Jaakko Hämäläinen

Selaimen renderöinnin optimointi

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikan koulutusohjelma

Insinöörityö

25.4.2016

Metropolia

Tekijä(t) Otsikko Sivumäärä Aika	Jaakko Hämäläinen Selaimen renderöinnin optimointi 41 sivua 25.4.2016
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Lehtori Simo Silander Vanhempi ohjelmistoinsinööri Alex Latvala

Insinöörityön tavoitteena on kuvata selainten renderöintimoottorien toimintaan web-kehittäjälle kannalta tärkeistä näkökulmista sekä esittää web-sivuston suorituskyvyn optimointiprosessia. Työssä esitellään renderöintimoottorin toimintaa eri vaiheissa HTML-dokumentin tulkitsemisesta elementtien näyttöruudulle renderöintiin. Lisäksi työssä esitellään joitain sivuston suorituskyvyn kannalta tyypillisiä pullonkauloja tyylimäärittelyissä sekä kuvataan näiden pullonkaulojen optimointiprosessit. Työssä kuvatut optimointiprosessit on esitetty käyttämällä hyväksi Google Chrome -selaimen devTools-työkaluja.

Insinöörityössä kuvataan modernille web-kehittäjälle tärkeitä tietoja renderöintimoottorin toiminnasta, tavoiteruudunpäivitysnopeuden saavuttamisesta sekä heikosti suoriutuvien sivustojen optimoinnista.

Tyypillisenä optimointiesimerkkinä työssä esitellään renderöintimoottoria kuormittavien animaatioiden optimointi CSS3:ssa esiteltyä transform: translate-funktiota hyväksi käyttäen. Toisenä esimerkkinä esitellään staattisesti asemoidun, renderöintimoottoria kuormittavan, taustakuvan optimointiprosessi.

Insinöörityössä esitellään nauhotteita optimoimattomien ja optimoitujen esimerkkisivustojen toiminnasta. Näistä nauhotteista voitiin havaita, miten suuri merkitys tietyillä tyylimäärittelyillä on sivuston suorituskyvyn näkökulmasta erityisesti vähätehoisilla laitteilla.

Avainsanat	Selaimen renderöintimoottori, suorituskyky, animaatio

Author(s) Title Number of Pages Date	Jaakko Hämäläinen Browser Rendering Optimization 41 pages 25 April 2016
Degree	Bachelor of Engineering
Degree Programme	Information and Communications Technology
Specialisation option	Software
Instructor(s)	Simo Silander, Senior Lecturer Alex Latvala, Senior Software Engineer

The goal of this thesis is to describe how browser rendering engines work from a web developer's point of view and to demonstrate the performance optimization process of a website. The thesis explains different phases of browser rendering engine actions from parsing the HTML document to rendering elements on the viewport and describes usual performance bottlenecks in style definitions and optimizing them. The optimization processes are explained using the Google Chrome devTools-toolkit.

The thesis also describes important information about rendering engine and its actions, acquiring the target frame rate and optimizing websites with weak performance.

Optimizing rendering engine burdening website animations with CSS3-announced transform: translate-function is demonstrated as a typical optimization example in the thesis. Another example demonstrates the process of optimizing a staticly positioned, rendering engine burdening background image.

Thesis demonstrates Google Chrome devTools-recordings of optimized and unoptimized website actions. The causality of unoptimized style definitions to browser rendering engine performance can be verified analyzing these recordings.

Keywords	Browser Rendering Engine, Performance, Animation

Sisällys

Lyhenteet

CSS	Cascading Style Sheets. CSS on Tyypillisesti WWW-dokumenttien elementtien ulkoasuun ja esitystapaan tarkoitettu kuvauskieli.
CSSOM	CSS Object Model. CSSOM on puurakenteinen tietorakenne, jonka renderöintimoottori rakentaa www-dokumentin elementteihin liittyvistä CSS-määrittelyistä.
DOM	Document Object Model. DOM on puurakenteinen tietorakenne, jonka renderöintimoottori rakentaa www-dokumentin elementeistä.
FPS	Frames Per Second. Ruudunpäivitystaajuus.
GPU	Graphics Processing Unit. Grafiikkayksikkö.
HTML	Hypertext Markup Language. HTML on kuvauskieli, jolla www-sivut on tyypillisesti kirjoitettu.
LESS	LESS on dynaaminen kuvauskieli. joka laajentaa CSS-kielen syntaksia. Kielellä kirjoitetut dokumentit käännetään tyypillisesti CSS-syntaksin mukaiseksi.

1 Johdanto

Tässä insinöörityössä kuvataan selaimen renderöintimoottorien toimintaa yleisesti sekä erityisesti silmällä pitäen keskimääräisen päätelaitteen näytön virkistystaajuutta vastaavan ruudunpäivitystaajuuden saavuttamista. Lisäksi insinöörityössä esitellään websivustoille tavanomaisia tyylimäärittelyjä, joiden renderöintimoottorille aiheuttamat prosessit estävät tavoiteruudunpäivitystaajuuden saavuttamisen sekä kuvaillaan keinot vastaavien toiminnallisuuksien saavuttamiseksi tinkimättä tavoiteruudunpäivitystaajuudesta.

Insinöörityöstä lukemisesta on etua ensisijaisesti moderneille web-kehittäjille, jotka ovat kiinnostuneita saavuttamaan visuaalisesti näyttäviä sekä vaivattomasti myös vähätehoisilla päätelaitteilla toimivia sivustoja sekä laajentamaan tietoaan selaimen renderöintimoottorin toiminnasta. Insinöörityö kuvaa lukijalle metodeja optimoida heikosti renderöityvä sivusto tavoiteruudunpäivitystaajuuden saavuttamiseksi.

Insinöörityössä esiteltyjen optimointiesimerkkien tueksi on luotu AngularJS-sovellus, jonka näkymissä esitetään läpikäytävät optimointiesimerkit. Sovellukselle on perustettu julkinen Git-versionhallinta-säilö, jonka osoite on https://bitbucket.org/jellyface/rendering-performance-demo.

Insinöörityön tilaaja on Tieto CEM (Custom Experience Management) ja työn tavoitteena tilaajan näkökulmasta on olla osana minun henkilökohtaista taitojen kehittämisprosessiani.

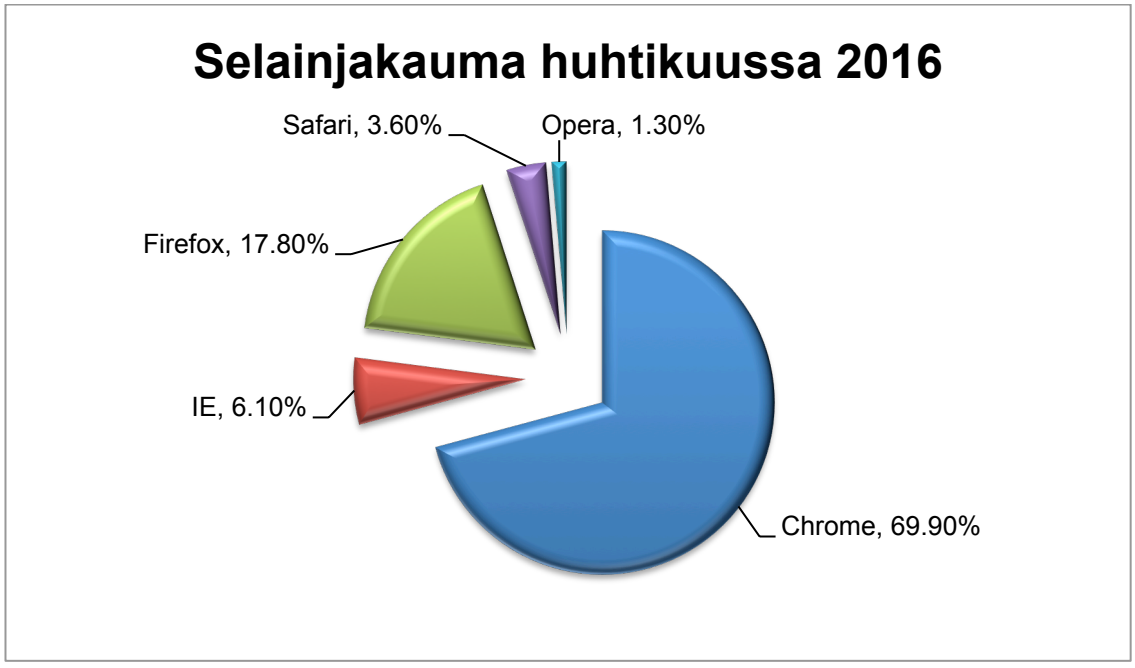
Työssä esiteltävät Chrome devTools-työkalut sekä optimointiprosessit, joissa niitä käytetään hyväksi, ovat versioriippuvaisia. Työssä on kuvattu Chrome 47 -version toimintaa. Myöhemmissä versioissa voi olla eroavaisuuksia toiminnallisuuksissa ja termeissä.

2 Selaimet ja selainmoottorit

2.1 Selaimet nyt

Tämän hetken käytetyimmät selaimet ovat Google Chrome, Mozilla Firefox, Internet Explorer (Microsoft) ja Safari (Apple). Näistä selvästi suosituin selain tällä hetkellä on Chrome, joka dominoi markkinoita 69,9 % käyttäjäosuudella. Firefoxin suosio oli huipussaan 47,9 % osuudella heinäkuussa 2009, jolloin ensimmäisestä stabiilista Chrome-julkaisusta oli kulunut 10 kuukautta. Tästä eteenpäin Chromen osuus on tasaisesti kasvanut Internet Explorerin ja Firefoxin osuuksien pienentyessä. Ennen tätä Internet Explorerin osuus oli parhaimmillaan 88,0 % maaliskuussa 2003. Firefox alkoi kasvattamaan suosiotaan sen kustannuksella aina ensimmäisestä virallisesta julkaisustaan asti marraskuussa 2004.

Edellä mainittujen lisäksi tällä hetkellä aktiivisessa käytössä on pääasiassa Mac OS -käyttöjärjestelmän mukaan paketoitu Applen kehittämä Safari-selain 3,6 %:n käyttäjäosuudella sekä norjalaisen Opera Softwaren kehittämä Opera-selain 1,3 % osuudella. Käyttöjakauma yleisimpien selainten välillä on esitetty kuvassa 1. (1.)



Kuva 1. Käyttöjakauma yleisimpien selainten välillä huhtikuussa 2016-04-10

2.2 Selainten kehityssyklit

Kiihtyvällä vauhdilla kehittyvä web-teknologia edellyttää, että myös nettiselaimet tukevat esimerkiksi JavaScriptin uusimpia ominaisuuksia. Moderneimmat, lyhyin väliajoin päivittyvät selaimet sisältävät jo toteutukset sellaisille JavaScript- ja CSS-ominaisuuksille, jotka on määritelty vasta luonnosteluvaiheessa olevissa määritelmissä. Toteutukset kaikista moderneimmille JavaScript- sekä CSS-ominaisuuksille löytyvät pääsääntöisesti vain Chromesta ja Firefoxista.

Chrome-selainta kehittävä Google siirtyi heinäkuussa 2010 aiemmasta hitaammasta julkaisuaikataulusta sykliin, jossa uuteen viralliseen versioon implementoitavat uudistukset käyvät läpi julkaisuhaaraputken kolmen eri haaran läpi ennen kuin ne lisätään viralliseen versioon. Uusia ominaisuuksia kehitetään ensin canary-haarassa, josta ominaisuudet tapauskohtaisesti joko hylätään tai otetaan jatkokehitykseen dev-haaraan kuudennen viikon alussa. dev-haarassa ominaisuuksia ja stabiliteettia jatkokehitetään edelleen ja samoin kuudennen kehitysviikon alkaessa ominaisuudet joko nostetaan beta-haaraan tai hylätään. Kuusi viikkoa tästä ja ominaisuudet julkaistaan virallisessa Chrome-versiossa 16 viikon kuluttua alkuperäisen kehittämisen alkamisesta. Koska kehitystä tehdään jokaisessa haarassa samanaikaisesti eri tiimien toimesta, Chromesta pyritään julkaisemaan virallinen versio aina kuuden viikon välein.

Firefox-selainta julkaiseva Mozilla ilmoitti vuoden 2011 alussa adoptoivansa hyvin paljon Chromen sykliä muistuttavan 16-viikkoisen kehityssyklin alkaen Firefox 4-version julkaisupäivästä. Kuten Google, myös Mozilla kehittää selaintaan neljässä eri haarassa, joista uudet ominaisuudet nostetaan aina seuraavaan haaraan ja lopulta viralliseen versioon 16 viikon päästä. Taulukossa 1 on esitettynä Chromen ja Firefoxin kehitysaikataulu

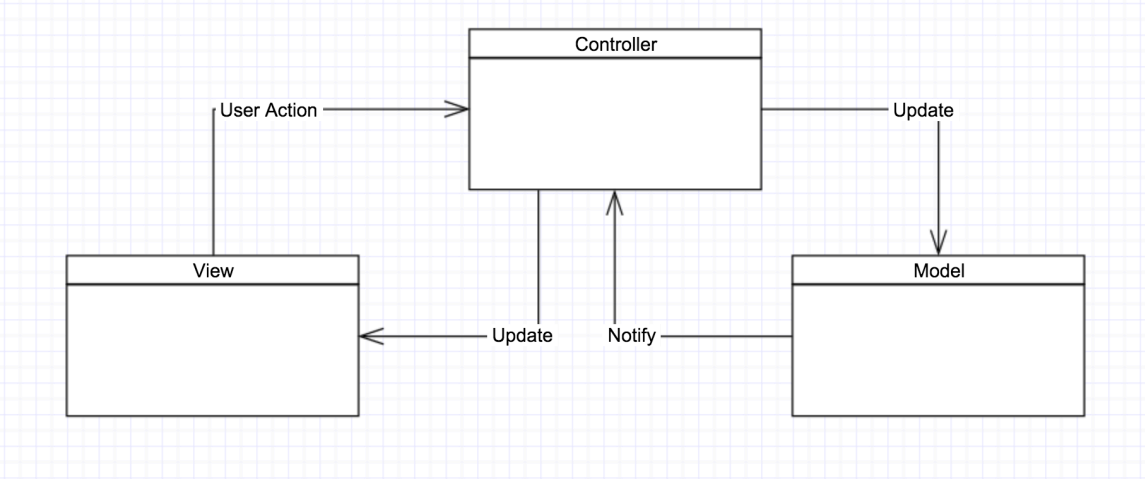
Taulukko 1. Selaimien kehityssyklit

Aika	Branch name (Chrome)	Branch name (Firefox)
Viikko 1-5	Canary	Nightly
Viikko 6-10	Dev	Aurora
Viikko 11-15	Beta	Beta
Viikko 16	Release	Release

Chromen ja Firefoxin lisäksi myös muut selainkehittäjät ovat adoptoineet aiempaa tiheämmät kehityssyklit.

2.3 Selaimen toiminta

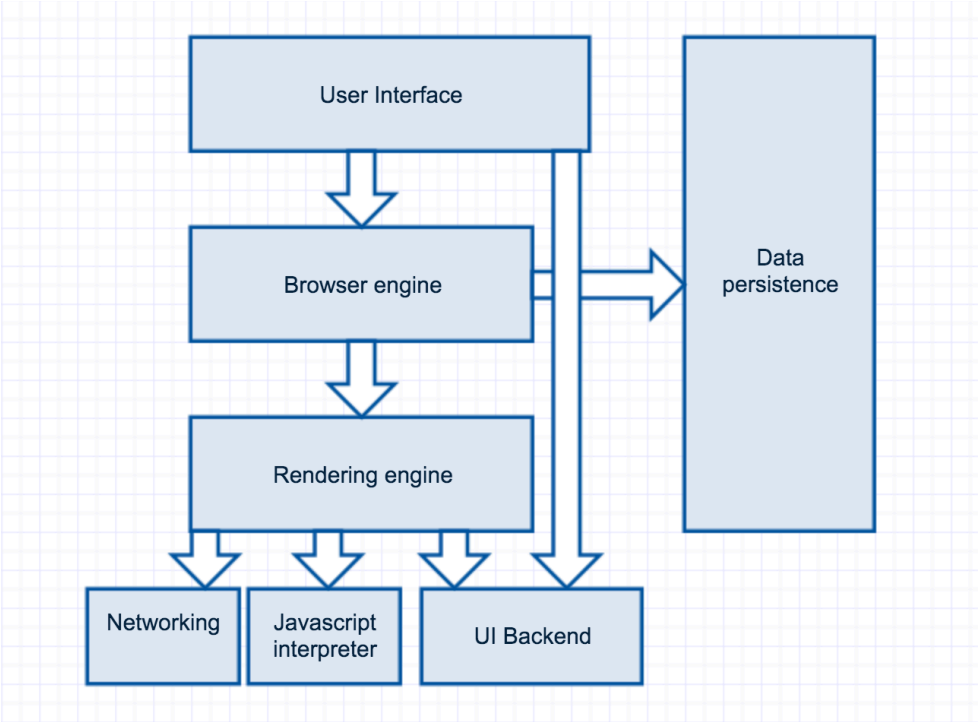
Jokaisen web-selaimen graafisen käyttöliittymän taustalla toimii kokoelma moduuleja, joista kukin vastaa omista selaimen toiminnalle kriittisistä osa-alueistaan. Nämä moduulit ovat selainmoottori, renderöintimoottori, networking-moduuli, JavaScript-tulkki, UI Backend -moduuli ja Data Persistence -moduuli. Selainmoottorin rooli selainta käytettäessä muistuttaa MVC-mallin (kuva 2) mukaista kontrolleria.



Kuva 2. Model-View-Controller-malli

Selainmoottorin tehtävänä on toimia viestinvälittäjänä graafisen käyttöliittymän ja itse renderöintimoottorin välillä.

Lisäksi selainmoottori toimii kontrollerina Data Persistence -moduulille, jossa säilytetään selaimeen tallennetut evästeet sekä HTML5-spesifikaatiossa esitelty Local Storage. Renderöintimoottori (Rendering Engine) puolestaan kommunikoi verkkoliikenteestä vastaavan networking-moduulin, JavaScript-skriptien tulkkaamisesta vastaavan JavaScript interpreter-moduulin sekä joidenkin HTML-elementtien, kuten lomakekenttien, oletusulkoasusta vastaavan UI Backend -moduulin kanssa. Kuva 3 havainnollistaa moduulien suhteita toisiinsa. (2.)



Kuva 3. Selaimen komponentit

Koska tämän työn tarkoitus on kuvata web-sivujen ja hybridimobiiliapplikaatioiden optimointia älypuhelimille, tableteille ja muille laitteille, joiden laskentateho on huomattavasti nykyaikaisia työpöytäkoneita ja kannettavia tietokoneita heikompi, käsitellään seuraavassa luvussa erityisesti selaimen renderöintimoottorin toimintaa.

3 Selainmoottorit

3.1 WebKit

WebKit on Applen kehittämä selainmoottori, jota Safari-selain käyttää. Apple käynnisti WebKit-selainmoottorin kehityksen alun perin kesällä 2001 Linux-järjestelmäympäristöihin suunnatun avoimen lähdekoodin ohjelmistopaketin KDE:n layout-moottori KHTML:n erillisenä kehityshaarana. Vuoden 2003 alussa Apple julkisti Safari-selaimensa, joka käytti WebKit-moottoria. Ohjelmistoprojektien kehityshaaroille ominaiseen tapaan projektien kehitystyön tuloksia pyrittiin kehityksen aikana yhdistämään KHTML:n ja WebKit:n välillä, mutta projektien erilaisista kehityssuunnista ja tavoitteita johtuen tämä osoittautui haasteelliseksi.

Kesällä 2005 Apple lisensoi WebKit-moottorin avoimen lähdekoodin lisenssillä, mikä mahdollisti muille selainkehittäjille WebKit-moottorin käyttöönoton. Googlen kehittämä Chrome-selain käyttikin Applen kehittämää moottoria selainmoottorinaan ensimmäisestä versiostaan lähtien aina kevääseen 2013 asti, jolloin Google perusti WebKit-moottoriin perustuvan kehityshaaran omalle Blink-selainmoottorilleen.

3.2 Blink

Blink-moottorin kehitys käynnistettiin siksi, että Google koki WebKitin sisältävän liikaa Safari-selaimeen liittyvää koodipohjaa. Google halusi toteuttaa abstraktimman implementaation WebKitistä, jossa ei olisi yksittäiseen selaimeen kohdentuvaa koodia. Tämä on se trendi, joka ohjaa Googlen muutakin ohjelmistokehitystä. Blink-moottori oli käytössä ensimmäisen kerran kesäkuussa 2013 julkaistussa Chrome 28:n Linux-versiossa, joka saapui muutamaa viikkoa myöhemmin myös Mac Os- sekä Windows-käyttöjärjestelmälle.

3.3 Presto

Opera-selaimen kehittäjä norjalainen Opera Software käytti selaimessaan aina vuoteen 2013 asti itse kehittämäänsä Presto-selainmoottoria. Ensimmäisen kerran Presto oli käytössä Opera 7 -versiossa, joka julkaistiin tammikuussa 2003. Helmikuussa 2013 Opera ilmoitti siirtyvänsä käyttämään vielä siinä vaiheessa Googlen ja Applen yhdessä kehittämää WebKit-moottoria ja pian sen jälkeen Googlen julkistaessa tiedon Blink-moottorin kehityksestä ilmoitti käyttävänsä jatkossa Googlen kehittämää selainmoottoria.

3.4 Gecko

Firefox-selaimen kehityksestä vastaava avoimen lähdekoodin ohjelmistoja julkaiseva Mozilla on alusta asti käyttänyt itse kehittämäänsä Gecko-moottoria paitsi selaimessaan, myös muissa tuotteissaan, kuten Thunderbird-sähköpostiohjelmassaan. Gecko-moottorin kehitys alkoi, kun Netscape Navigator -selainta kehittänyt Netscape irrotti Communicator-nimiseksi muuttuneen selaimensa 4-version selainmoottorista lisensoi-

mattomat komponentit ja julkaisi ne Mozilla-nimisenä projektina avoimen lähdekoodin lisenssin alla maaliskuussa 1998.

Lokakuussa 1998 Netscape ilmoitti seuraavaksi julkaistavan Navigator-version käyttävän Gecko-moottoria. Navigator 5 oli tällöin jo alpha-vaiheessa, mutta päätös moottorin vaihtamisesta johti siihen, ettei Navigator 5-versio saanut ikinä virallista julkaisua, vaan kiireellisellä aikataululla kehitetty ensimmäinen Gecko-pohjainen Navigator nimettiin suoraan 6-versioksi. Netscape Navigator 6 sai kuitenkin kritiikkiä keskeneräisyydestään, ja selaimen käyttäjäkunta alkoi pienentyä. Tämä johtui merkittävin osin päätöksestä käyttää vielä keskeneräistä sekä epävakaata Gecko-moottoria. Myöhemmät Navigator-julkaisut olivat huomattavasti vakaampia, mutta eivät onnistuneet voittamaan muiden selaimien pariin siirtyneitä käyttäjiä enää puolelleen.

2003 Netscapen omistaja AOL päätti katkaista Navigator-selaimen kehityksen, ja suuri osa sen parissa työskennelleistä kehittäjistä siirtyi Mozilla-projektia hallinnoivan Mozilla Organizationille jatkokehittämään Gecko-moottoria sekä kehittämään uutta siihen pohjautuvaa selainta, Mozilla Firefoxia. Firefoxin ensimmäinen virallinen julkaisu tapahtui marraskuussa 2004, ja se kasvoi yhdeksi käytetyimmistä selaimista ennen Chromen julkaisua ja on edelleen maailman toiseksi käytetyin selain.

3.5 EdgeHTML ja Trident

Microsoftin kehittämä Trident on ollut sen kehittämän Internet Explorerin moottori 1997 julkaistusta IE4-versiosta lähtien. Tätä edeltävissä versioissa käytettiin pohjana Spyglassin kehittämää Mosaic-selainta. Internet Explorer oli ylivoimaisesti suosituin selain julkaisustaan lähtien aina 2000-luvun loppupuolelle asti, osittain sen vuoksi, että se tuli maailman suosituimman käyttöjärjestelmän, Microsoft Windowsin mukana.

Trident on kuitenkin saanut Mozilla Firefoxin ja varsinkin Google Chromen julkaisun jälkeen merkittävästi kritiikkiä käyttäjiltä sen hitaudesta sekä erityisesti web-kehittäjiltä sen vuoksi, ettei se noudattanut web-standardeja tai tukenut uusia web-teknologioita lainkaan samoissa määrin kuin kilpailijat Firefox ja Chrome. Puutteellinen web-standardien tuki johti siihen, että web-kehittäjien oli huomioitava Internet Explorerin käyttäjät mm. käyttämällä erillisiä vendor-prefixejä tyylimäärittelyissään sekä ylimääräi-

siä fallback-määrittelyitä niissä tapauksissa, joissa toivottu määrittely ei ollut millään tavalla toteutettavissa kyseisellä IE-versiolla.

IE6-versiota seuraaviin versioihin Microsoft keskittyikin huomattavasti enemmän kehittämään Trident-moottoria siihen suuntaan, että se tukisi samoja web-teknologioita kuin sen kilpailijat. IE7 ja IE8 tukivatkin kumpikin huomattavasti edeltäjäänsä paremmin uusia teknologioita, mutta vasta 2011 julkaistu IE9 tuki lähes kaikkia samoja teknologioita kuin sen aikaiset kilpailijansa sekä pärjäsi niille jossain määrin selaintesteissä.

Vaikka IE9:n julkaisu karkoittikin web-kehittäjien pahimmat antipatiat Microsoftia kohtaan, Microsoftin selain aiheutti edelleen jonkin verran päänvaivaa kehittäjille. Iso osa Microsoftin selaimen käyttäjistä eivät omatoimisesti päivittäneet selaintaan, eikä Microsoft erityisen aktiivisesti kehottanut heitä tekemäänkään päivitystä. Tämä johti siihen, että vielä yli vuosi IE9:n julkaisun jälkeen, IE7 ja IE8 käyttivät yhteenlaskettuna enemmän käyttäjiä kuin uutta IE9-versiota, mikä pakotti web-kehittäjiä huomioimaan Microsoftin vanhojen selainten puutteet vielä pitkään niiden korjaantumisesta uusissa versioissa. (3.)

Vuonna 2015 heinäkuussa Microsoft julkaisi uuden Windows 10 -käyttöjärjestelmän, jonka oletusselain oli uusi Microsoft Edge. Edge käyttää uutta EdgeHTML-moottoria, joka on kehitetty Tridentin pohjalta ja joka keskittyy entistä enemmän karsimaan Tridentin nykyisin tarpeettomia ominaisuuksia sekä kilpailemaan Firefoxin ja Chromen kanssa web-standardien tuen ja nopeuden suhteen. Sen vastaanotto onkin ollut hyvä sekä käyttäjien että web-kehittäjien keskuudessa ja erityisesti sen JavaScript-tulkin suorituskykyä suhteessa vanhoihin Trident-selaimiin on kehuttu.

4 Selaimen renderöinti

4.1 Teoria renderöinnin optimoinnin takana

Nykypäivänä nettiselainten käyttäjät ovat tottuneet edellyttämään vierailemiltaan sivuilta, että ne toimivat sujuvasti ja tottelevat käyttäjän antamia syötteitä johdonmukaisesti ja tehokkaasti. Näitä asioita edellytetään myös silloin, kun itse laite, jolla surffaillaan on taskukokoinen älypuhelin tai tabletti, jolla on ongelmia tarjota sellaista laskentatehoa, jota modernin web-sivuston käyttäminen voi vaatia. Tätä silmällä pitäen nykyaikaisen

web-kehittäjän on syytä tuntea selaimen renderöintimoottorin toiminta yleisesti sekä erilaisten CSS-ominaisuuksien vaikutus sivuston suorituskykyyn.

Se, minkä käyttäjä kokee sivuston sulavana ja pehmeänä toimintana, on itse asiassa ruudunpäivitystaajuuden yhtenäisyys sivustolla navigoitaessa. Käyttäjän aiheuttaessa muutoksia renderöityyn layoutiin esimerkiksi vierittämällä näyttöä ylös tai alas tai käynnistämällä animaatioita, renderöintimoottori joutuu prosessoimaan käyttäjän aiheuttamat muutokset ja laskemaan sivuston komponenttien sijainnit uudestaan. Tämä edellyttää laskentaa, jonka tulee tapahtua tietyssä ajassa, jotta ruudunpäivitystaajuus säilyisi yhtenäisenä.

Yleiseksi tavoiteruudunpäivitystaajuudeksi on muodostunut 60 kuvaa sekunnissa, joka pohjautuu siihen, että suurin osa nykypäivän laitteiden näytöistä toimii 60 hertsin virkistystaajuudella. Tämä ei ole aina mahdollista, jolloin voidaan tyytyä myös 30 hertsin ruudunpäivitystaajuuteen, jota pidetään myös varsin riittävänä hyvän sulavuuden takaamiseksi, kunhan ruudunpäivitystaajuus pysyy systemaattisesti samana eikä heittele merkittävästi kumpaankaan suuntaan.

Tavoiteruudunpäivitystaajuuden ollessa esimerkiksi 60 hertsiä, voidaan laskea aika, joka renderöintimoottorilla on jokaisen kuvan luomiseen. Tätä aikaa kutsutaan usein nimellä *Frame Budget.*

$$\frac{1s}{60Hz} = 16.66 \ldots ms$$

Renderöintimoottorin tulisi siis rakentaa uusi kuva esitettäväksi ~16.67 millisekunnin välein, jotta ruudunpäivitystaajuus pysyisi optimaalisena. Modernin webkehittäjän tulisi pyrkiä kehittämään ohjelmistoprojektit ruudunpäivitysnopeuden suhteen sellaiseen tilaan, missä niin kaikki ruudulla näkyvät piirto-operaatiot kuin sovelluslogiikassa sijaitsevat funktiokutsut kaikki suoriutuvat alle määritellyn *Frame Budgetin*. Säännölliset *Frame Budgetin* ylittämiset voi havaita web-sivustoa selatessa tai sen animaatioita seuratessa käyttäjää epämiellyttävänä ”värinänä”, jolloin ruudulla päivittyvät objektit esimerkiksi siirtyvät tai muuttavat muotoaan hidastuvalla nopeudella. Tätä ilmiötä kutsutaan web-optimoinnista puhuttaessa normaalisti nimillä *jitter* tai *jank*.

4.2 Miksi tasainen ruudunpäivitystaajuus on tärkeää?

Nopean ruudunpäivitystaajuuden saavuttaminen ei itsessään ole välttämättä riittävä motivaattori web-sivuston refaktoroinnille, joten on syytä ymmärtää minkälaisia vaikutuksia sillä on sellaisiin mittareihin, joista jokaisen web-sivustoa kehittävän tulisi olla kiinnostunut.

Facebookilla työskentelevä Shane O'Sullivan kertoi Web-kehitykseen ja moderneihin web-teknologioihin keskittyvässä Edge Conference -keskustelupaneelissa 2013 Facebookin kokeilleen 20-30 miljoonan käyttäjän testiryhmällä, minkälaisia seurauksia sillä oli, kun he keinotekoisesti rajoittivat ruudunpäivitystaajuuden 60 kuvan päivitystaajuudesta kolmeenkymmeneen kuvaan sekunnissa iOS- ja Android-applikaatioissa ja totesivat sen romauttaneen applikaation sitouttavuuden käyttäjiin (4). Voidaan olettaa, että epätasaisesti esimerkiksi 15-30 kuvan päivitystaajuudella toimivan applikaation sitouttavuus olisi kärsinyt vielä merkittävämmin.

4.3 Renderöintimoottorin toiminta yleisesti

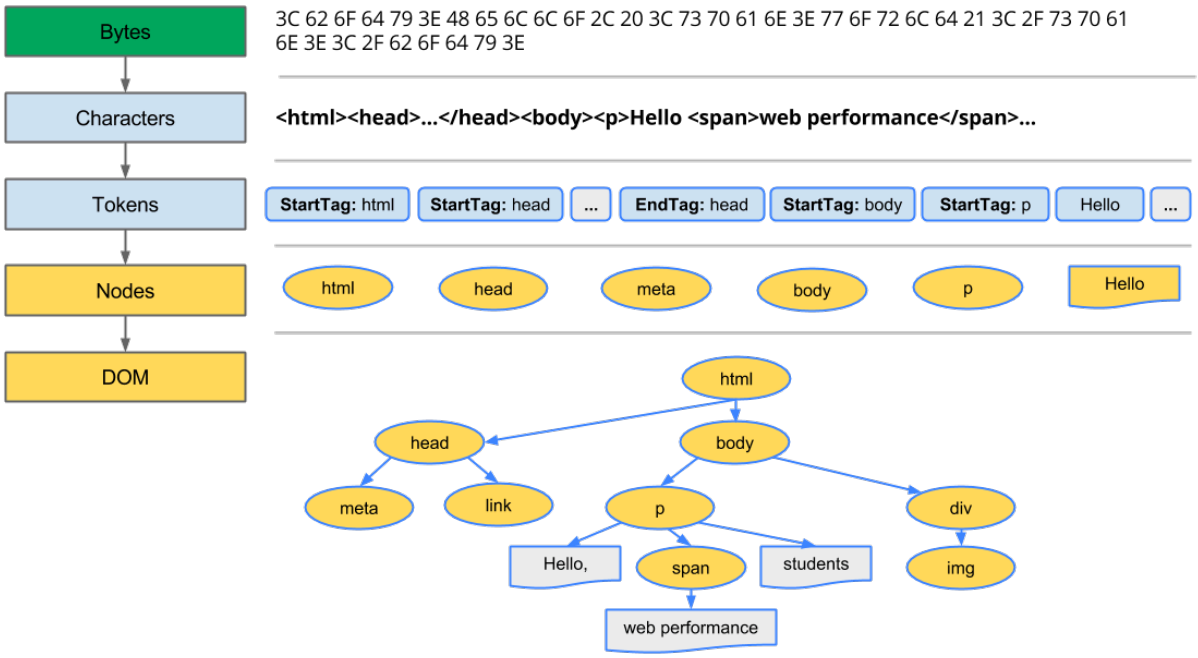
Kun käyttäjä syöttää nettiselaimen osoitekenttään URL-osoitteen ja painaa enter-nappia, selain käynnistää sekvenssin operaatioita. Verkkoliikenteestä vastaava Networking-moduuli noutaa palvelimen URL-osoitteesta palauttavan HTML-tiedoston ja renderöintimoottori alkaa jäsentämään sitä, välittäen Networking-moduulille viittaukset HTML-tiedoston käyttämiin ulkoisiin tiedostoihin niiden noutamiseksi. Kun HTML-tiedosto on jäsennetty, renderöintimoottori rakentaa sen perusteella rakenteeltaan puurakennetta muistuttavan DOM-objektin (Document object model) sekä ulkoisista tyylitiedostoista tai HTML-tiedoston sisällä esitetyistä tyyleistä CSSOM-objektin (CSS Object Model). Seuraavassa vaiheessa nämä kaksi objektia yhdistetään objektiksi (renderöintipuu), jonka perusteella renderöintimoottori asemoi sivuston komponentit (Layout) ja piirtää ne näytölle (Paint). Lisäksi selain tallentaa osia renderöintipuusta tasoiksi välimuistiin uudelleenpiirtojen optimoimiseksi (Composite).

4.4 DOM-puun luonti

Ensimmäinen operaatio, jonka renderöintimoottori suorittaa saadessaan selaimen networking-moduulilta html-tyyppistä sisältöä on lukea sisältö ja muuttaa se yksittäisistä tavuista (Bytes) merkeiksi (Characters) riippuen sisällössä määritellystä merkistökoodauksesta (Oletusarvoisesti UTF-8 HTML5-sivustolla). Seuraava esitys kuvaa, miten renderöintimoottori tulkitsee tavukoodin UTF-8-merkistökoodauksella HTML-syntaksilla html-elementin aloittavaksi komennoksi.

$$3C\ 68\ 74\ 6D\ \ 6C\ 3E \rightarrow < html >$$

Tämän operaation jälkeen Tulkittava data on jo ihmissilmälle ymmärrettävämpää, mutta renderöintimoottorin tulee vielä tunnistaa datasta html-syntaksin mukaiset elementit sekä muodostaa niistä tunnisteet (token / Lexing), joiden perusteella DOM-puun solmujen rakentaminen voidaan aloittaa. Sen lisäksi, että datasta luotujen tunnisteiden perusteella luodaan DOM-puun jokainen solmu, tunnisteiden esiintymisjärjestyksen perusteella voidaan päätellä myös solmujen väliset suhteet toisiinsa eli itse puun rakenne. Kuva 4 esittää renderöintimoottorin toimintaa tavukoodin tulkitsemisesta aina DOM-puun luontiin.

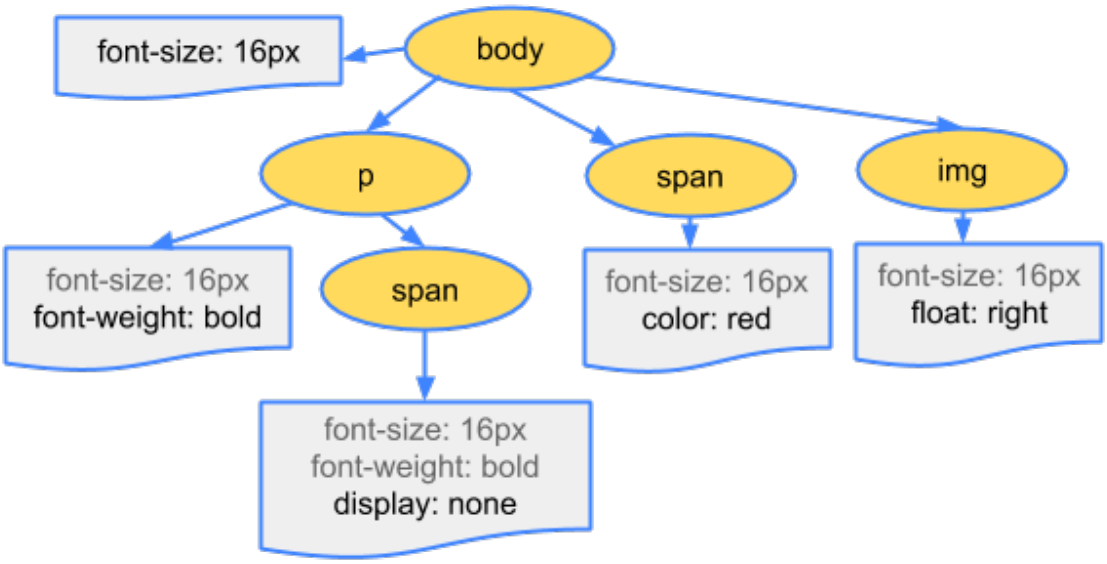


Kuva 4. Document-Object Model-puun luonti (5)

Tämän prosessin jälkeen renderöintimoottori on saanut luotua DOM-objektin kokonaisuudessaan, mutta itse DOM-objektissa ei oteta vielä kantaa eri solmujen tyyleihin, jotka tulee olla selvitettynä ennen kuin renderöintimoottori voi laskea eri elementtien asemointeja, joten seuraavaksi renderöintimoottori käsittelee dokumentin sisällä esitetyt ja sen sisältä viitatun ulkoisen tekstimäärittelytiedoston CSS-tyylit sekä luo näistä tyylimäärittelyistä rakenteeltaan DOM-puuta muistuttavan objektin, CSSOM-puun.

4.5 CSSOM-puun luonti

Tässä vaiheessa DOM-puu on valmis, mutta sivuston piirtäminen sekä esittäminen käyttäjälle on mahdotonta, sillä DOM-puu ei sisällä minkäänlaista tietoa elementtien asetteluista sekä muista mahdollisista tyylimäärittelyistä. Tyylien esittämiseksi renderöintimoottorin tulee ensin rakentaa CSSOM-puu (CSS Object-model) prosessoimalla html-dokumentin sisällä sekä siinä viitattujen ulkoisten tyylitiedostojen sisällä esitetyt tyylimäärittelyt tavalla, joka muistuttaa monilta osin edellisessä kappaleessa DOM-puun luomisen yhteydessä käytyä prosessia. Kuva 5 kuvaa CSSOM-puun luomisprosessia tyylimäärittelyistä.



Kuva 5. CSSOM-puun luonti (5)

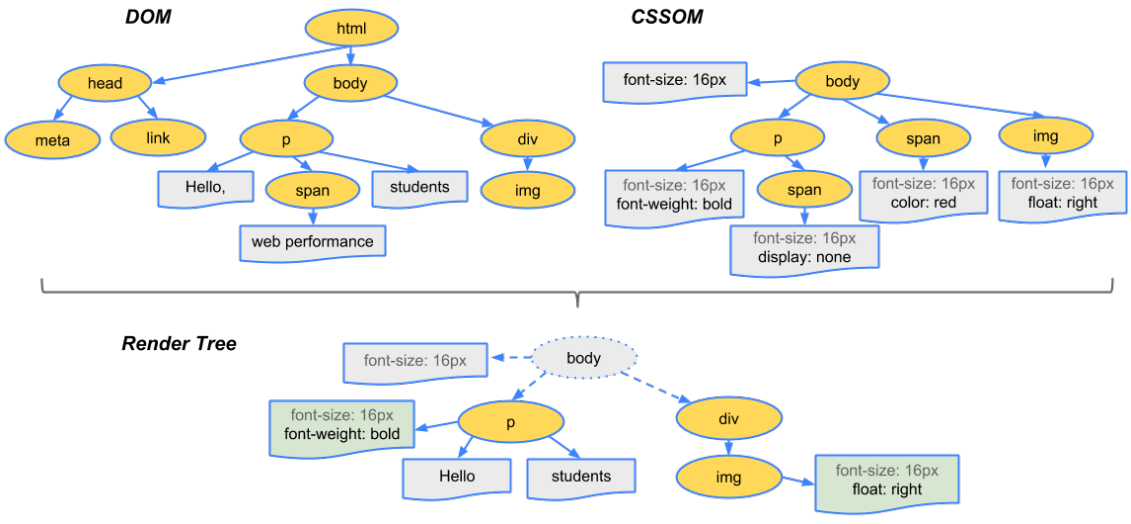
Ensimmäisessä vaiheessa renderöintimoottorin on käännettävä esimerkiksi html-dokumentin head-osiossa esitellyn tyylimäärittelytiedoston tyylimäärittelyt yksittäisistä tavuista määritellyn merkistökoodauksen mukaisesti merkeiksi ja tunnistaa merkkijonot

CSS-syntaksin mukaisesti. Luoduista tunnisteista rakennetaan CSSOM-puun solmut, joiden relaatiot pystytään myös tulkitsemaan syntaksin mukaisten selektoreiden avulla.

4.6 Renderöintipuun luonti

Ennen kuin renderöintimoottori voi laskea html-dokumentin eri elementtien ulottuvuuksia ja sijainteja selaimen näkymässä, sen on yhdistettävä DOM-puu ja CSSOM-puu yhdeksi renderöintipuuksi, jonka sisällöstä on yksiselitteisesti tulkittavissa kunkin elementin sisältö sekä elementtikohtaiset tyyliominaisuudet. Puiden yhdistäminen renderöintipuuksi on suhteellisen suoraviivainen prosessi, jossa on kuitenkin muutamia mielenkiintoisia yksityiskohtia.

Huomiota herättävin ero renderöintipuun ja DOM-puun välillä on se, etteivät DOM-puun kaikki solmut päädy renderöintipuuhun. DOM-puu sisältää solmuja, jotka eivät ole ikinä nähtävissä näytöllä. Esimerkiksi head-elementti ja sen sisältämät meta- ja script-tagit jätetään HTML-spesifikaatioiden mukaisesti aina piirtämättä. Samoin sellaiset DOM-puun solmut, jotka piiloitetaan esimerkiksi display:none-määritelmällä, jäävät renderöintipuun ulkopuolelle. Kuva 6 havainnollistaa prosessia, jossa renderöintimoottori luo DOM-puun sekä CSSOM-puun perusteella renderöintipuun



Kuva 6. Renderöintipuun luonti (6)

Seuraavaksi niille DOM-puun solmuille, jotka päätyvät renderöintipuuhun, kullekin liitetään CSSOM-puusta vastaavat tyylimäärittelyt. Kukin solmu käydään läpi ja siihen

kohdistuvien CSS-valitsimien sisällä määritetyt ominaisuudet liitetään renderöintipuun solmuun. Tämän lisäksi ne määrittelyt, jotka oletusarvoisesti periytyvät lapsielementeille, lisätään solmun lapsien tyylimäärittelyihin. Viimeisessä vaiheessa kuhunkin solmuun liitetään itse sisältö, jonka jälkeen renderöintipuu sisältää DOM-puusta poimittuna kaikki HTML-dokumentissa esitetyt elementit sekä CSSOM-puusta poimittuna kaikki niihin selektoituvat tyylimäärittelyt.

4.7 Layout-prosessi

Kun renderöintipuu on valmis, renderöintimoottori tietää kaikkien näkyvien elementtien hierarkian ja tyylimäärittelyt, mutta elementtien tarkat sijainnit näytöllä sekä niiden mittasuhteet suhteessa näyttöikkunaan ovat vielä laskematta. Elementtien käsittely tapahtuu vaiheittain alkaen hierarkisesti ensimmäisestä solmusta, body-elementistä. Tämän jälkeen kukin solmu lapsisolmuineen käydään läpi järjestyksessä vasemmalta oikealle ja ylhäältä alas, ja niiden mittasuhteet ja sijainnit lasketaan.

Järjestys on siinä mielessä looginen, että myöhemmin käytävien solmujen ominaisuudet eivät pääsääntöisesti vaikuta edellisten solmujen asemointiin. Iso osa solmun asemointiin ja mittoihin vaikuttavista css-ominaisuuksista on usein suhteellisia sen isäsolmun asemointiin ja mittoihin, jolloin ne on oltava laskettuina ennen kuin lapsisolmun ominaisuuksia voidaan laskea.

4.8 Paint-prosessi

Layout-prosessin jälkeen renderöintimoottori tietää kaikkien renderöintipuun solmujen sijainnin ja mittasuhteet, mutta näytölle piirtämistä varten sen on vielä prosessoitava solmuista luotavien laatikkomallien muut ulkoasuun liittyvät ominaisuudet. Renderöintimoottorin on tämän jälkeen piirrettävä pikseli kerrallaan kunkin ruudulla näytettävän elementin sisältö ruudulle.

Renderöintimoottorin täytyy laskea jokaisen yksittäisen pikselin väri sen perusteella, minkälaisia CSS-ominaisuuksia on elementillä, jonka osa pikseli on. Tässä laskennassa on huomioitava ominaisuuksia, kuten background-color, border-radius, box-shadow, sekä erilaiset reunanpehmennykseen vaikuttavat ominaisuudet. Laskenta on toistetta-

va jokaiselle piirrettävälle pikselille, mikä tekee paint-prosessista laskennallisesti suhteellisen raskaan operaation.

Renderöintimoottorille raskaat operaatiot ovat ongelma ruudunpäivityksen kannalta, sillä laskennan tavoiteruudunpäivitystaajuuden (60 Hertsiä) saavuttamiseksi selaimen olisi kyettävä prosessoimaan uusi kuva 16,6 millisekunnin välein. Tämän vuoksi paint-prosessien minimoimisella web-sivun ajon aikana on keskeinen rooli siinä, miten tavoiteruudunpäivitystaajuus voidaan saavuttaa tinkimättä sivuston visuaalisesta ilmeestä tai animaatioiden näyttävyydestä.

4.9 Composite-prosessi

Renderöintimoottori suorittaa ennaltaehkäiseviä toimintoja paint-prosessien minimoimiseksi ajon aikana tallentamalla tarvittaessa osia renderöintipuusta välimuistiin, ja rakentamalla niistä *tasoja (layer)*, joita koostamalla renderöintimoottori voi piirtää kuvan. Muutokset omassa tasossaan olevassa elementissä eivät aiheuta uusia paint-prosesseja muissa elementeissä, minkä vuoksi selaimet usein nostavat omiin tasoihin sellaiset elementit, joiden uskotaan muuttuvan usein esimerkiksi animaatioiden johdosta.

Chromen parissa työskentelevä kehittäjä Shawn Singh määritteli toukokuussa 2013 pitämässään esityksessä *Compositing in Blink and WebKit* ajonaikaisen paint-prosessin kahteen eri osaan, *repaint* ja *redraw*. Singhin määrittelemä repaint-prosessi vastaa pitkälti ensimmäisen latauksen yhteydessä tehtävää Paint-prosessia, joka kaikkine osaprosesseineen kuormittaa laitteistoa merkittävästi. Redraw-prosessissa sen sijaan haetaan omalla tasollaan oleva muuttunut komponentti välimuistista. Tämän jälkeen Redraw-prosessissa suoritetaan Paint-prosessin osaprosessit vain elementissä muuttuneiden ominaisuuksien osalta sekä renderöidään elementti uudestaan. Singh toteaa puheessaan redraw-prosessin olevan merkittävästi suorituskyvyltään kustannustehokkaampi verrattuna repaint-prosessiin. (7.)

Tiettyjen ehtojen täyttyessä osa sivuston elementeistä korotetaan omille tasoilleen. Animoitujen elementtien korottaminen on suositeltavaa erityisesti laitteilla, joiden laskentateho on rajallinen, kuten älypuhelimilla ja tableteilla. Toisaalta liiallinen elementtien korottaminen omille tasoilleen voi johtaa grafiikkayksikön muistin loppumiseen, joka

romauttaa paitsi sivuston ruudunpäivitystaajuuden, koko laitteen toiminnan. Tämän vuoksi modernin web-kehittäjän on tunnettava selaimien kompositiologiikka sekä CSS-määritykset, joiden perusteella selaimet korottavat komponentteja omille tasoilleen.

4.10 Stacking Context ja z-index

Asemoitaessa tietyllä tavalla HTML-elementit voivat myös pinoutua toistensa päälle osittain tai kokonaan. Pinoutuneiden elementtien asemaan z-akselin suuntaisesti eli syvyyssuunnassa voidaan vaikuttaa CSS-määrityksellä *z-index*. Oletusarvoisesti eli kun z-index = *auto*, elementit pinoutuvat siinä järjestyksessä, kun ne esiintyvät HTML-dokumentissa, viimeiseksi esiteltyjen piirtyessä viimeisenä. Muuttamalla vakioarvoa voidaan luoda elementille ja sen lapsille muista sivun elementeistä riippumaton ympäristö pinoutumiselle, jota kutsutaan nimellä *Stacking context*. (8.)

4.11 Dirty Bit -mekanismi

Nykyaikaiset web-sivut saattavat muuttua visuaalisesti hyvinkin radikaalisti ajon aikana esimerkiksi käyttäjän syötteestä, mikä edellyttää selaimelta layout-prosessin ja paint-prosessin uudelleenajoa. Modernit renderöintimoottorit ovat optimoituja tätä silmällä pitäen ja pyrkivät oletusarvoisesti kohdistamaan nämä prosessit vain niille sivuston elementeille, jotka ovat muuttuneet edellisen ruudunpäivityksen jälkeen.

Koko renderöintipuuhun kohdistuvaa layout- tai paint-prosessia kutsutaan globaaliksi prosessiksi ja vain muuttuneisiin elementteihin kohdistuvaa inkrementaaliksi prosessiksi. Renderöintimoottori pitää yllä informaatiota uuden layout-prosessin tai paint-prosessin tarpeellisuudesta Dirty Bit -systeemillä. Kullakin yksittäisellä renderöidyllä elementillä on oma bittinsä, joka kuvaa sitä, onko kyseiselle elementille ja sen lapsielementeille sekä isäelementille tarpeellista suorittaa layout- tai paint-prosessi uudelleen.

Elementin koon tai sijainnin muuttaminen CSS-määrityksillä edellyttää myös sisarelementtien uudelleenasemointia, joten kyseisen elementin isäelementin yhteyteen merkitään tieto siitä, että sille ja sen lapsielementeille on ajettava sekä layout- että paint-prosessi. Vaihtoehtoisesti muutettaessa ominaisuutta, joka ei vaikuta elementin sijain-

tiin tai kokoon, kuten taustaväriä tai fontin kokoa, voidaan layout-prosessi jättää suorittamatta ja suorittaa elementille ja sen lapsielementeille vain paint-prosessi.

Dirty Bit -mekanismilla on keskeinen rooli laiteresursseja kuormittavien layout- ja paint-prosessien määrän minimoimisella, mutta sen optimaalista toimintaa edellyttää se, että DOM-puu sekä CSSOM-puu ovat rakenteeltaan loogisia sekä semanttisesti korrekteja.

5 Renderöinnin mittaaminen

5.1 Mittaustyökalut

Jotta selaimen renderöintimoottorin toimintaa voidaan optimoida, sen toimintaa on kyettävä mittaamaan jollain validilla tavalla. Nykyään suosituimpina työkaluina renderöintimoottorin toiminnan mittaamiseen toimivat käytettävän selaimen integroidut kehittäjätyökalut. Pääosin eri selaimien työkalut muistuttavat toiminnallisuudeltaan hyvin paljon toisiaan ja niiden käyttö on niin samanlaista, että yhden selaimen kehittäjätyökaluja aiemmin käyttänyt kehittäjä kykenee käyttämään myös minkä tahansa muun selaimen kehittäjätyökaluja.

Eri selainkehittäjät kutsuvat omia työkalujaan eri nimillä. Osittain näiden yksittäisten työkalujen nimet ovat yleistyneet tarkoittamaan myös selaimen kehittäjätyökaluja käsitteenä yleisesti.

Taulukko 2. Eri selaimien kehittäjätyökalut

Selain	Kehittäjätyökalu	Tyyppi
Google Chrome	devTools	Integroitu
Apple Safari	Web Inspector	Integroitu
Mozilla Firefox	Developer Tools	Integroitu
Mozilla Firefox	Firebug	Plugin
Opera	Dragonfly	Plugin
Microsoft IE / Edge	Developer Tools	Integroitu

Integroitujen työkalujen lisäksi suosittuja on vastaavaan tarkoitukseen kehitetyt selainliitännäiset, kuten Firefoxille kehitetty Firebug. Lisäksi eri selaimille on saatavilla liitännäisiä, jotka tuovat lisäominaisuuksia integroituun kehittäjätyökaluun. Näistä liitännäisistä erityisen suosittuja ovat sellaiset, jotka lisäävät kehittäjätyökaluihin jonkin front-

end-ohjelmistokehyksen, kuten AngularJS:n, Ember.js:n tai react.js:n, kannalta keskeisiä lisäominaisuuksia.

Tässä työssä kuvataan jatkossa kehitystyökalun käyttö renderöintimoottorin toiminnan optimoimiseksi käyttäen Google Chromen devTools-työkalua, joka on tämän insinöörityön kirjoittajan mielestä ominaisuuksiltaan kehittynein ja dokumentaatioltaan paras näistä työkaluista.

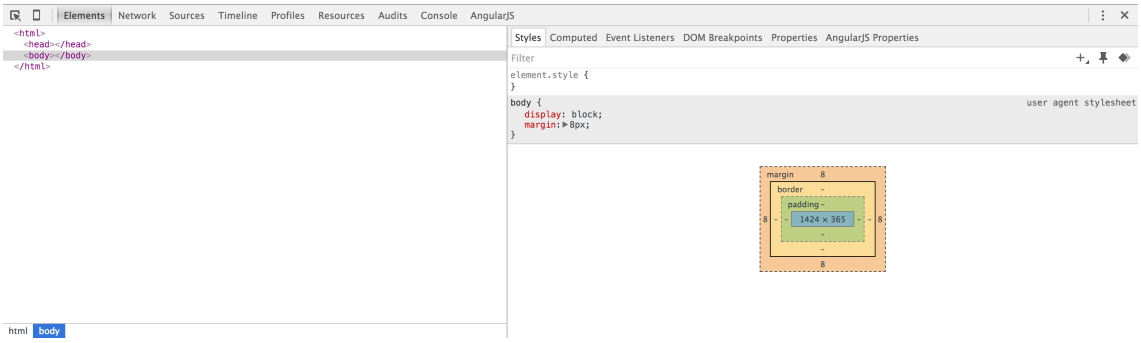
5.2 devToolsin käyttöönotto ja toiminnallisuudet

Google Chromen devTools voidaan käynnistää valitsemalla Chromen valikosta *More Tools*-osion alta Developer tools -vaihtoehdon. Vaihtoehtoisesti devTools voidaan käynnistää web-sivulla klikkaamalla hiiren oikealla näppäimellä mitä tahansa web-sivun elementtiä ja valitsemalla *Inspect Element* tai näppäimistön pikakomennolla. Pikakomento devToolsin avaamiseksi on windows-käyttöjärjestelmällä F12 tai Ctrl + Shift + I ja Mac Os-käyttöjärjestelmällä Cmd + Opt + I.

Chrome devToolsin toiminnallisuudet on jaettu kahdeksalle eri välilehdelle, joista selaimen renderöinnin optimoinnin kannalta keskeisimpiä ovat Elements- ja Timeline-välilehdet. Näiden lisäksi web-kehittäjän useimmiten tarvitsemia työkaluja ovat Console-välilehdeltä löytyvä JavaScript-konsoli sekä Network-välilehdeltä löytyvä listaus sivuston suorittamista ulkoisista resurssikutsuista esimerkiksi sivuston käyttämään REST-rajapintaan.

5.2.1 Elements-välilehti

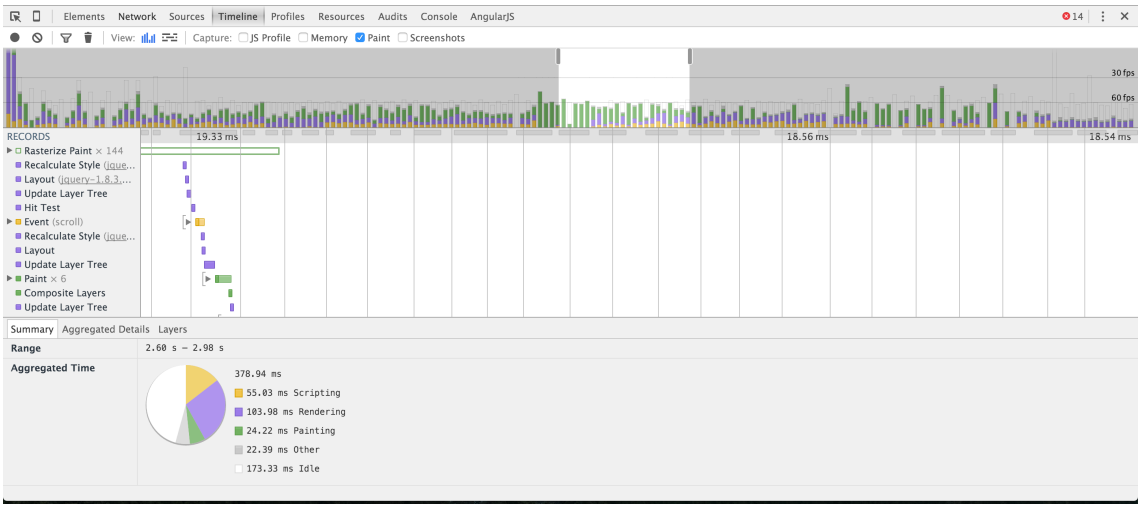
Elements-välilehdellä (kuva 7) käyttäjä voi tutkia reaaliaikaista representaatiota DOM-puusta HTML-muodossa ikkunan vasemmassa reunassa sekä tästä representaatiosta valitun HTML-elementin tyyliominaisuuksia ikkunan oikeassa reunassa. Kummassakin ikkunassa voi tehdä muutoksia, mikä mahdollistaa nopean prototyyppi-henkisen kehitystyön, muuttamatta sivuston alkuperäistä lähdekoodia lainkaan.



Kuva 7. devToolsin Elements-välilehti

5.2.2 Timeline-välilehti

Timeline-välilehdellä (kuva 8) voidaan analysoida selaimen sisäisiä prosesseja ruudunpäivityksien välillä suorituskyvyn kannalta. Analysoinnin tavoitteena on kohdentaa verkkosivuston suorituskykyä heikentäviä ominaisuuksia. Analysointia ennen käyttäjän on nauhoitettava nauhoite, jonka aikana sivustoa selataan tai sille annetaan käyttäjäsyötteitä. Nauhoittaessa on syytä pyrkiä simuloimaan sivuston käyttöä mahdollisimman monipuolisesti analysoidessa sivuston yleistä suorituskykyä tai vaihtoehtoisesti simuloida tietoisesti vain sivuston yksittäisen komponentin käyttöä halutessa rajata analysointia sivuston tiettyyn toiminnallisuuteen.



Kuva 8. devToolsin Timeline-välilehti

Nauhoitus käynnistetään ja lopetetaan Timeline-välilehden yläpalkin ensimmäisestä ikonista vasemmalta katsottuna. Nauhoituksen päätyttyä devTools analysoi nauhoitetun datan ja lisää yhteenvedon siitä Timeline-välilehden näkymään. Työkalupalkista on valittavissa olevista moodeista renderöinnin mittaamisen kannalta tärkein on Frames-moodi, jonka aktivointi tapahtuu view-katerogian ensimmäisestä painikkeesta.

Frames-moodissa työkalupalkin alapuolella Timeline-näkymässä on pylväsdiagrammi, joka kuvaa renderöintimoottorin operaatioiden suoritusnopeutta. Diagrammissa olevat poikittaiset apuviivat osoittavat ruudunpäivityksen osalta kriittisiä raja-arvoja. Alemman apuviivan alittavat operaatiot suoriutuvat riittävän nopeasti 60 hertsin ruudunpäivitystaajuutta silmällä pitäen eli kukin alle 16,67 millisekunnissa, ja ylemmän apuviivan alittavat operaatiot riittävän nopeasti 30 hertsin ruudunpäivitystaajuutta silmällä pitäen eli kukin alle 33,34 millisekunnissa.

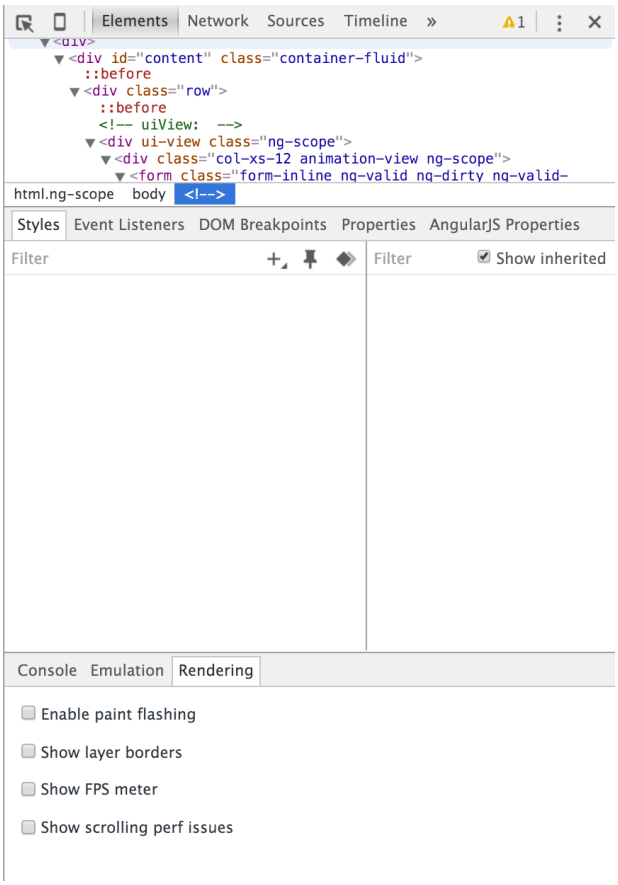
Pylväsdiagrammin pylväät koostuvat eri värisistä komponenteista, joista kukin kuvaa selaimen eri prosessin ajallista kustannusta.(9) (10)

- Sininen – Networking sekä HTML-tiedostojen käsittely
- Keltainen – JavaScript-skriptien suoritus
- Purppura –Layout-prosessi
- Vihreä – Paint-, ja Composite-prosessi
- Harmaa – Prosessoria kuormittava prosessi, jota devTools ei pysty yksilöimään
- Läpinäkyvä – Selain on toimettomana.

Heikosti optimoiduista web-sivustoista nauhoitetut nauhoitteet osoittavat usein sen, ettei renderöintimoottori kykene 60 hertsin päivitystaajuuteen pääsääntöisesti liian pitkien Layout- tai Paint-prosessien vuoksi. Tämä ilmenee Timeline-välilehden pylväsdiagrammissa erityisen korkeina, pääasiallisesti vihreistä tai purppuroista pylväistä.

5.2.3 devToolsin renderöintiasetukset

Google Chromen devTools sisältää muutamia päällekytkettäviä lisäominaisuuksia, jotka auttavat raskaalta vaikuttavan sivuston ongelmakohtien löytämisessä. Ominaisuudet kytketään päälle devTools-konsolin Rendering-välilehdeltä (kuva 9).



Kuva 9. Chrome devTools-työkalun Rendering-välilehti

Enable paint flashing -valinta asettaa Google Chrome -selaimen tilaan, jossa kaikki sivustolla suoritettavat paint-prosessit esitetään visuaalisesti piirtämällä vihreän läpikuultavan laatikon sen alueen päälle, jossa selain suorittaa paint-prosessia. Valinnan aktivointi on yksi helpoimmista tavoista yksilöidä sivuston suorituskykyä hidastavien paint-prosessien käynnistävät elementit.

Show layer borders -valinta asettaa Chromen tilaan, jossa Chrome piirtää oranssilla värillä reunukset niille elementeille, joiden piirtämisen se korottaa sillä hetkellä omaksi tasokseen, jonka käyttäytyminen ei vaikuta muihin sivuston elementteihin. Show layer borders- valinta piirtää lisäksi sinisellä värillä sivustolle ruudukon, joka kuvaa, miten Chrome jakaa sivustolla esitettävät suuret tasot komponenteiksi, joina ne välitetään piirrettäviksi. Ruudukolla ei ole juuri koskaan roolia paint-prosessien optimoinnissa, mutta kehittäjän on tarpeellista ymmärtää, ettei se kuvaa samaa asiaa kuin oranssit kehykset.

Show FPS meter valinnan kytkeminen piirtää sivuston oikeaan yläkulmaan pienen ikkunan, jossa kuvataan senhetkinen ruudunpäivitysnopeus sekä grafiikkayksikön senhetkinen muistinkäyttö. FPS meter -ikkunasta tietojen perusteella voidaan tehdä jossain määrin samoja johtopäätöksiä piirto-operaatioiden raskaudesta kuin Enable paint flashing -valinnan piirtämistä vihreistä laatikoista. Ruudunpäivitysnopeuden ollessa tasaisesti 60 ruutua sekunnissa voitaisiin päätyä johtopäätökseen, että sivuston animaatiot on optimoitu riittävälle tasolle, mutta esimerkiksi säännöllisesti muuttuvat lukemat grafiikkayksikön muistinkäyttöä kuvaavassa GPU memory -kohdassa sivuston animaatioiden ollessa käynnissä antavat ymmärtää, että renderöintimoottori suorittaa jatkuvasti paint-operaatioita.

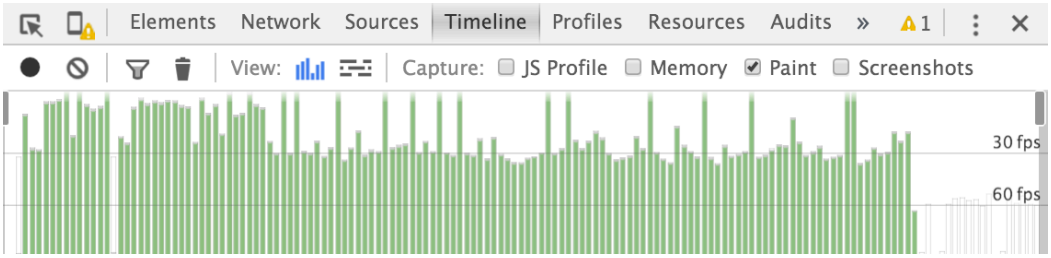
5.3 Renderöinnin optimointi devToolsilla

Web-sivuston renderöinnin optimointi voidaan jakaa vaiheisiin, jotka toistamalla voidaan optimoida mikä tahansa heikosti optimoitu sivusto. Seuraavissa luvuissa kuvataan mm. Google Chromen kehittäjien sekä muiden selaimen renderöintiin perehtyneiden asiantuntijoiden suositusta kyseisestä prosessista. (11; 12; 13.)

5.3.1 Ongelmia aiheuttavien elementtien kohdentaminen

Ensimmäisessä vaiheessa kehittäjä kartoittaa sivuston yleistä toiminnallisuutta laukaisemalla mahdollisia animaatioita sekä selailemalla sivustoa ylösalaisin. Renderöinnin optimoinnin suhteen kokeneempi kehittäjä saattaa jo tässä vaiheessa nähdä silmämääräisesti, mikäli sivuston renderöinti ei toimi riittävän sulavasti sekä missä mahdolliset ongelmakohdat ovat. Ongelmakohtien varmistamiseksi on aina järkevää tehdä devTool Timeline-nauhoite edellä mainituista toimenpiteistä sekä tarkastella tuloksia Frames-moodissa.

Nauhoitusta kuvaavan pylväsdiagrammin taustaväriltään vihreiden pylväiden ylittäessä tavoiteruudunpäivitysnopeuteen riittävän suoritusajan säännöllisesti, tiedetään ongelman aiheutuvan paint-prosessien liiallisesta määrästä. Timeline-nauhoitusten lisäksi tarkasteluvaiheessa voi olla hyödyllistä asettaa aiemmassa otsikossa esitetyt enable paint flashing-, sekä show layer borders -moodit päälle devToolsin rendering-välilehdeltä. Kuva 10 esittää timeline-nauhoitusta optimoimattomasta sivustosta, jonka ruudunpäivitysnopeus on heikko.



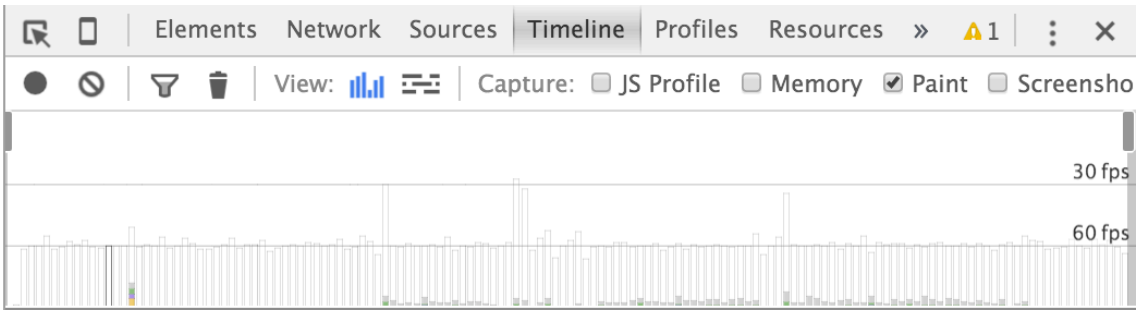
Kuva 10. Optimoimattoman sivuston nauhoite

5.3.2 Kohdennettujen elementtien verifiointi

Edellä mainittujen toimenpiteiden jälkeen kehittäjällä on yleensä jo suhteellisen hyvä käsitys siitä, mikä elementti aiheuttaa sivuston heikon suorituskyvyn. Seuraavassa vaiheessa tämä on hyvä todentaa poistamalla yksi kerrallaan sellainen elementti renderöintipuusta, jonka epäillään aiheuttavan jatkuvia paint-prosesseja.

Tämä tapahtuu helpoiten valitsemalla kyseinen elementti devToolsin Elements-välilehdeltä sekä asettamalla sen tyyliominaisuuksiin joko display:none- tai visibility:hidden-määritys. Pääasiallisena erona näiden kahden välillä on se, että display:none poistaa elementin myös DOM-puusta, jolloin sen viemää tilaa näytöllä ei oteta huomioon myöskään layout-prosessissa. Käyttämällä visibility:hidden-määritelmää voidaan olla varmoja, että sivuston layout ei muutu elementin poistamisen johdosta, sillä elementti poistuu vain renderöintipuusta. Chrome devToolsissa visibility:hidden-määritys voidaan asettaa helposti pikanäppäimellä (H) elementin ollessa valittuna Elements-välilehdellä.

Kun elementti on poistettu renderöintipuusta, voidaan uuden timeline-nauhoitteen tietojen perusteella päätellä, paraniko sivuston performanssi kyseisen elementin ollessa poissa renderöintipuusta. Kuvassa 11 on nähtävissä timeline-nauhoite sivustosta, jonka ruudunpäivitysnopeus on kiitettävä. Timeline-välilehden pylväsdiagrammi osoittaa paint-prosessien vähentyneen merkittävästi sekä renderöintimoottorin suoriutuvan riittävän hyvin tehtävistään 60 ruudun ruudunpäivitysnopeuden saavuttamiseksi (Läpinäkyvät pylväät kuvaavat aikaa, jonka selain viettää toimettomana ruudunpäivitysten välillä).



Kuva 11. Optimoidun sivuston nauhoite

5.3.3 Toimenpiteet ongelmallisten elementtien tunnistamisen jälkeen

Renderöintiperformanssiin vaikuttavan elementin kohdentaminen on merkittävä osa optimointiprosessia, mutta ei vielä itsessään nopeuta renderöintimoottorin toimintaa silloin, kun elementti on renderöintipuussa. Seuraavaksi kehittäjän on tunnistettava kyseisen elementin tyyliominaisuuksista renderöintimoottorille tarpeettomia paint-prosesseja aiheuttavat ominaisuudet sekä korvattava ne mahdollisuuksien mukaan ominaisuuksilla, joiden muuttaminen ei aiheuta tarpeettomia paint-prosesseja.

Paint-prosesseja aiheuttavien tyyliominaisuuksien tunnistaminen vaatii kehittäjältä perehtyneisyyttä asiaan, mutta myös kokemattomampi kehittäjä pystyy tähän käyttämällä hyväkseen web-sivustoja, jotka listaavat CSS-ominaisuuksia sekä niiden laukaisemia prosesseja renderöintimoottorissa. Google Chrome kehittäjien Paul Lewisin ja Paul Irishin kirjoittamassa artikkelissa (14) listataan Google Chromen osalta tärkeimpiä sekä viitataan Google Sheets-dokumenttiin, jossa määrityksiä on listattu kattavammin (15). Lisäksi Paul Lewis ylläpitää CSS Triggers-sivustoa (16), jossa myös ylläpidetään listaa CSS-ominaisuuksista, sekä niiden laukaisemista prosesseista .

Seuraavassa pääluvussa esitellään muutamia yleisimpiä tarpeettomia paint-prosesseja aiheuttavia CSS-ominaisuuksia sekä tapoja saavuttaa sama funktionaalisuus vaihtamalla ominaisuudet sellaisiin, jotka mahdollistavat tavoitetun ruudunpäivitysnopeuden saavuttamisen myös vähätehoisilla laitteilla.

6 Tapaustutkimus

Web-sivut ovat renderöinnin optimoinnin kannalta aina yksilöllisiä, eivätkä yhden web-sivuston optimointiin vaadittavat toimenpiteet ole ikinä sanatarkasti replikoitavissa toisen web-sivuston optimoimiseksi. Sivustoissa on kuitenkin joitain tyypillisiä usein toistuvia ominaisuuksia, joiden optimointi vähentää renderöintimoottorin paint-prosesseja merkittävästi. Tässä pääluvussa esitellään tavanomaisimpia web-sivuston optimointikohteita. Pääluvun tueksi on perustettu esimerkkiprojekti, jonka versionhallinta on osoitteessa https://bitbucket.org/jellyface/rendering-performance-demo . Projektin tarkoitus on havainnollistaa web-kehittäjälle tyypillisiä optimointikohteita sekä tarjota materiaalia timeline-nauhoitusten tulosten analysointiin. Projektin kussakin näkymässä esitellään yksi tyypillinen optimointikohde sekä html form-elementti, jonka valitsimista voi vaihtaa optimoitavien elementtien tyylit optimoituun muotoon. Web-kehittäjä voi projektissa helposti testata optimoitujen tyyliominaisuuksien vaikutusta paint-prosessien määrään tekemällä timeline-nauhoituksia sekä optimoimattomilla että optimoiduilla tyyliominaisuuksilla.

6.1 Projektin teknologiat sekä ympäristön asennus

Esimerkkiprojekti on toteutettu käyttäen AngularJS-kehystä sekä Bootstrap-kehystä. Projektia ajetaan Node.js-palvelinympäristössä (17) Grunt-työkalulla (18) ja pakettienhallintaan käytetään grunt-liitännäisten osalta NPM-pakettimanageria (19) sekä frontend-kirjastojen osalta bower-pakettimanageria (20). Projektin lataamiseen voidaan käyttää git-versionhallintatyökalua (21). Koko projektiympäristön voi pystyttää suorittamalla alla listatut tehtävät.

- Node.js-ympäristön asennus (https://nodejs.org)
- Git-versionhallintatyökalun asennus (https://git-scm.com)
- `npm install -g grunt-cli bower`
- `cd rendering-performance-demo`
- `bower install`
- `npm install`
- `grunt serve`

- Siirry sivustolle http://localhost:9001 valitsemallasi selaimella

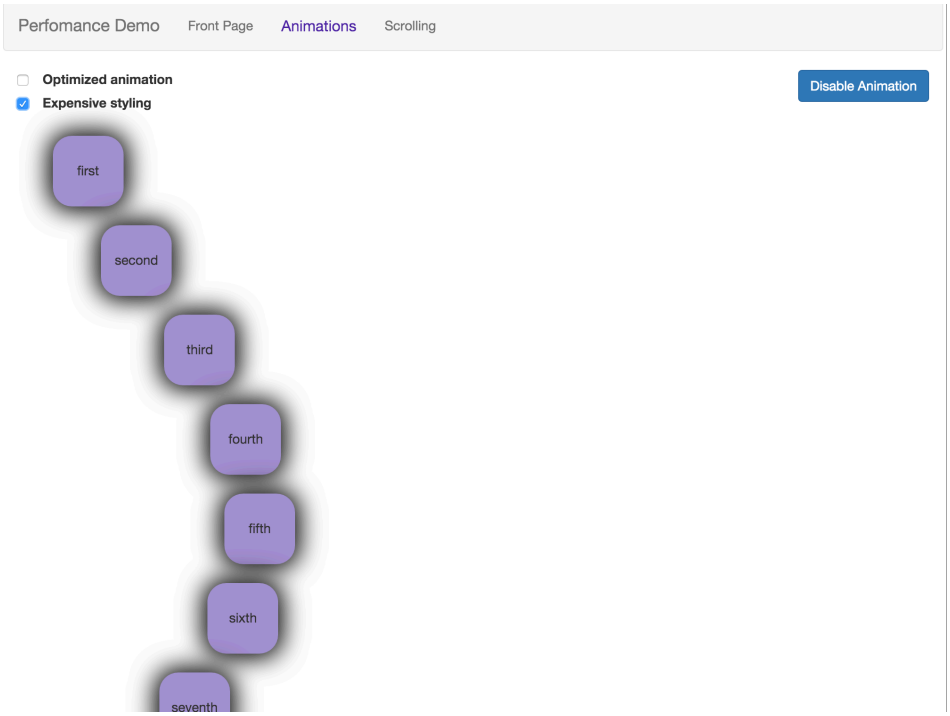
6.2 Animaatioiden optimointi

Tämän alaluvun ensisijaisena tarkoituksena on opastaa modernia web-kehittäjää käyttämään CSS-animaatioissaan mahdollisuuksien mukaan ominaisuuksia, jotka mahdollistavat sulavan ruudunpäivitysnopeuden, vähätehoiset laitteet mukaan lukien, kaikilla päätelaitteilla. Yksinkertaisin tapa saavuttaa tavoiteruudunpäivitysnopeus animoinnin yhteydessä on pyrkiä käyttämään luvussa listattuja transform-ominaisuuuksia, jotka eivät aiheuta jatkuvia paint-prosesseja renderöintimoottorissa.

Tarpeettomista paint-prosesseista erityisesti vähätehoisille laitteille aiheutuvien ruudunpäivitysongelmien korostamiseksi tämän alaluvun kuvissa esiteltävät timeline-nauhoitukset on nauhoitettu Sony Xperia Z3 -puhelimella tehokkaan pöytätietokoneen tai kannettavan sijaan.

6.2.1 Ongelman kohdentaminen

Esimerkkiprojektin Animations-näkymässä (http://localhost:9001/#/animations) (Kuva 12) on nähtävissä kymmenen animoitua div-elementtiä sekä form-elementti, jonka kontrolleista animoitujen elementtien tyyliominaisuuksiin voi tehdä muutoksia.



Kuva 12. Optimoimaton Animations-näkymä

Div-elementit animoidaan käyttämällä CSS3:n animation-ominaisuutta sekä keyframes-syntaksia määrittämään elementtien asema animaation eri vaiheissa. Lisäksi elementeille annetaan animation-delay-ominaisuus jokaiselle elementille uniikilla arvolla, mikä saa elementit animoitumaan aaltoliikettä muistuttavassa syklissä. Optimized Animation-valinnan ollessa poistettuna elementit käyttävät kuvan 13 mukaista keyframes-animaatiota, joka manipuloi elementtien left-ominaisuutta, siirtäen niitä tietyn määrän oikealle dokumentissa riippuen animaation vaiheesta.

```
@keyframes unoptimized {
  0% {
    left: 0px;
  }
  50% {
    left: 200px;
  }
  100% {
    left: 0px;
  }
}
```

Kuva 13. Optimoimaton keyframes-animaatio

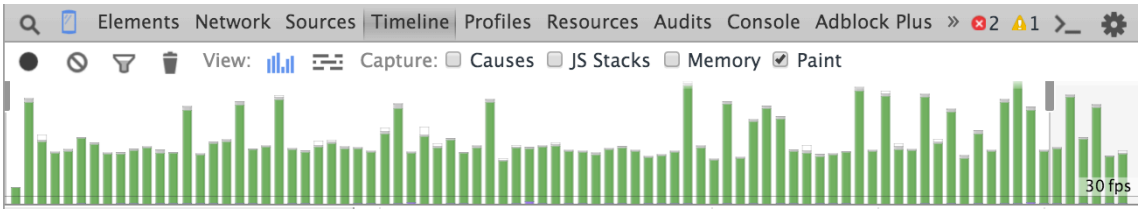
Tarkastellessa sivuston animaatiota vähätehoisella laitteella, kuten esimerkiksi älypuhelimella tai tabletilla, web-kehittäjä voi todennäköisesti jo silmämääräisesti havaita, että elementit eivät animoidu pehmeästi, vaan ne liikkuvat kuin värähdellen. Tällaiset havainnot viittaavat epätasaiseen ruudunpäivitysnopeuteen, joka alittaa todennäköisesti projektille asetetun tavoiteruudunpäivitysnopeuden.

Ruudunpäivitysnopeus putoaa entisestään, kun sivuston form-elementistä aktivoidaan expensive styling -valinta, joka asettaa animoitaville div-elementeille lisäksi kuvan 14 mukaiset määritykset, joiden käsittelyä pidetään laskentatehollisesti esimerkillisen raskaina operaatioina renderöintimoottorille. Border-radius-ominaisuudella voidaan pyöristää elementin reunoja ja box-shadow-ominaisuudella asettaa elementin reunoille varjostusta.

```
border-radius: 20px;
box-shadow: 0px 0px 40px 5px #000;
```

Kuva 14. Laskennallisesti raskaat tyylimäärittelyt

Aktivoimalla Chrome devToolsin Enable paint flashing -moodi, nauhoittamalla timeline-nauhoite sekä seuraamalla muita edellisessä pääluvussa esiteltyjä optimointiprosessin vaiheita voidaan todentaa, että renderöintimoottori joutuu jokaista renderöitävää ruutua varten suorittamaan paint-prosessin jokaiselle animoitavan div-elementin asemalla erikseen, mikä laskee ruudunpäivitysnopeuden tavoitenopeuden alapuolelle. Kuvassa 15 on esitettynä timeline-nauhoitus sivuston selailusta älypuhelimella animoinnin ollessa optimoimatonta sekä Expensive Styling -valinnan ollessa päällä.



Kuva 15. Timeline-nauhoite optimoimattomasta animaatiosta

Nauhoitteesta voidaan tulkita, ettei sivusto itse asiassa saavuta kyseisellä mobiililaitteella edes 30 ruudun ruudunpäivitysnopeutta. Yleisimmin tavoitenopeudeksi asetettava 60 ruudun ruudunpäivitysnopeus jää itse asiassa diagrammin skaalan ulkopuolelle,

sillä ruudunpäivitys ei kohonnut nauhoituksen aikana missään vaiheessa 30 ruudun päivitysnopeutta nopeammaksi.

6.2.2 Kohdennettujen elementtien optimointi

Kytkemällä elementtien animointi pois ”disable animation”-napista voidaan varmistua, että jatkuvat paint-operaatiot johtuvat ainoastaan animaatioista, tarkennettuna keyframes-animaation sisällä muuttuvista tyyliominaisuuksista. Tutkimalla aiemmassa luvussa esiteltyä CSS Triggers -sivustoa (16) voidaan todeta, että keyframes-animaation sisällä muuttuva left-ominaisuus käynnistää renderöintimoottorissa layout-prosessin, jonka seurauksena renderöintimoottorin on aina piirrettävä joitakin pikseleitä uudelleen, mikä tarkoittaa paint-prosessin käynnistymistä. Myös left-ominaisuutta elementin muilla sivuilla vastaavat top-, bottom-, ja right-ominaisuudet käynnistävät samat prosessit.

Sivuston suorituskyvyn nostamiseksi web-kehittäjän on kyettävä korvaamaan esimerkkitapauksessa left-ominaisuus vaihtoehtoisella ominaisuudella, pyrkien silti pitämään elementtien käyttäytyminen mahdollisuuksien mukaan täysin ennallaan. Tämä on mahdollista korvaamalla left-ominaisuus transform-ominaisuudella, joka esiteltiin ensimmäisen kerran CSS 3 -standardin marraskuussa 2009 julkaistussa työvedoksessa (22).

Chrome-kehittäjät Paul Lewis ja Paul Irish listaavat kirjoittamassaan artikkelissa (14) neljä tapaa käyttää transform-ominaisuutta elementtien animointiin, joilla elementin renderöinti voidaan korottaa erilliselle tasolleen renderöitäväksi. Korotettujen tasojen renderöintivastuu siirretään laitteen grafiikkayksikölle, minkä vuoksi näiden elementtien animointi on nopeata, eikä aiheuta paint-prosesseja animaation aikana.

- transform: translate()
- transform: scale()
- transform: rotation()
- transform: opacity()

Elementin asemoinnin animointiin näistä soveltuu transform-ominaisuuden translate-funktio, jolla voidaan korvata left-ominaisuudet esimerkkiprojektin keyframes-animaatiossa muuttaen se kuvan 16 mukaiseksi. Koska elementtien liike on alkuperäi-

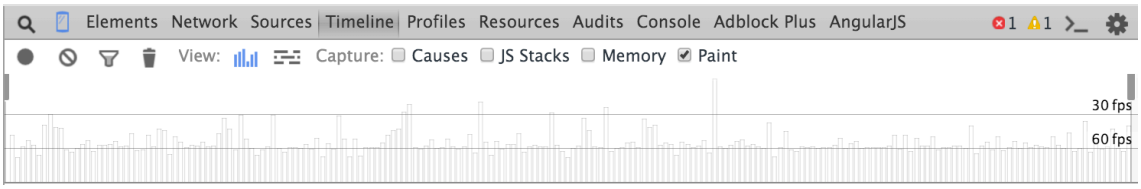
sessä animaatioissa sivuttaissuuntaista, voidaan translate-funktion sijaan käyttää translateX-funktiota, joka tarkentaa animoinnin x-akselin suuntaiseksi.

```
@keyframes optimized {
  0% {
    transform: translateX(0)
  }
  50% {
    transform: translateX(200px)
  }
  100% {
    transform: translateX(0px)
  }
}
```

Kuva 16. Optimoitu keyframes-määrittely

Esimerkkiprojektissa optimoitu keyframes-animaatio voidaan asettaa animoituihin elementteihin aktivoimalla Optimized Animation -valinta form-elementistä.

Nauhoittamalla timeline-nauhoite (kuva 17) sivulta Optimized animation-, sekä Expensive styling-valinnan ollessa päälle kytkettynä ja tarkastelemalla nauhoitetta voidaan todeta, että optimoitu animaatio ei enää laukaise jatkuvia paint-prosesseja renderöintimoottorissa. Läpinäkyvät pylväät kuvaavat aikaa, jonka renderöintimoottori viettää toimettamana ruudunpäivitysten välissä. Chrome-kehittäjä Paul Lewis avaa syitä ajoittaisille epäsäännöllisyyksille läpinäkyvien pylväiden sarjassa vastauksessaan Stack Overflow-sivustolla esitettyyn kysymykseen (23). Lewis myös kuvaa näiden epäsäännöllisyyksien vaikutusta sivuston käyttökokemukseen olemattomiksi.



Kuva 17. Timeline-nauhoite optimoidusta animaatiosta

Animaatioiden optimoinnin jälkeen sivusto pyörii sulavasti myös vähätehoisilla mobiililaitteilla tavoittaen toivotun 60 ruudun ruudunpäivitysnopeuden vaivattomasti.

6.3 Näytön vierityksen optimointi

Tämän alaluvun tarkoituksena on ohjata web-kehittäjää kiinnittämään huomiota sellaisiin renderöintimoottorin suorituskyvyn romauttaviin tyylimäärityksiin, joiden funktionaalisuus voidaan korvata jollain vähemmän paint-prosesseja laukaisevalla määrityksellä. Esimerkkiprojektin tapauksessa background-attachment:fixed-määritys korvataan sijoittamalla erilliseen div-elementtiin, joka asemoidaan position:fixed-määrityksellä vastaavan toiminnallisuuden tavoittamiseksi.

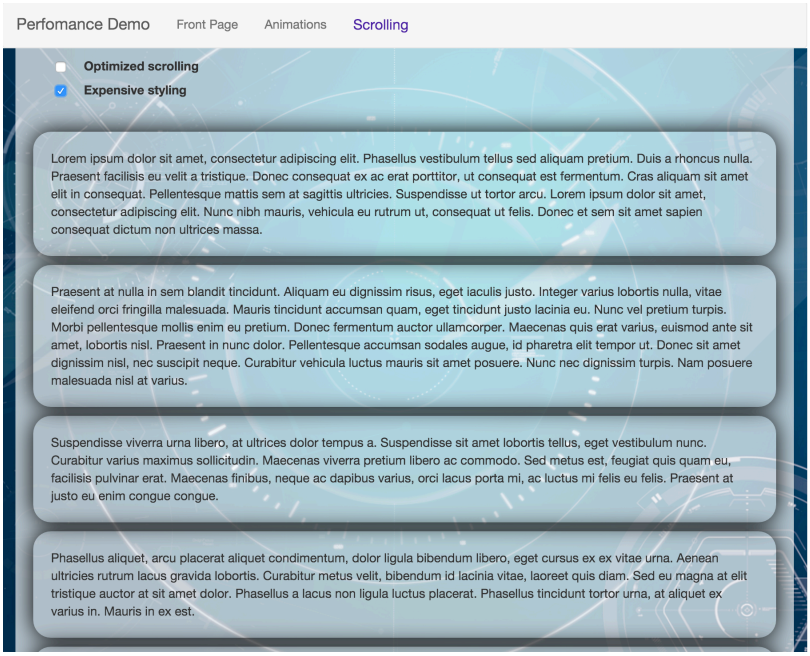
Toisena tarkoituksena on kuvata web-kehittäjälle koko ruudun kattavan *Paint Storm*-efektin optimointiprosessia. Paint-prosessien ollessa jatkuvasti koko ruudun kokoisia, ei yksittäisiä elementtejä piiloittamalla voida kohdentaa paint-prosesseja aiheuttavia elementtejä. Tällöin on järkevää yksilöidä ensin DOM-puun alin koko ruudun kokoinen elementti ja poistaa siltä yksitellen tyylimäärityksiä sekä analysoida näiden määritysten vaikutusta Enable paint flashing -asetuksen ollessa päällä.

Tässä alaluvussa esiteltävä staattisen taustakuvan optimointiongelma ei sovellu suurimmalle osalle tämän hetkisistä mobiiliselaimista (Opera Mini 8, Android Browser 47, Chrome for Android 49). Soveltumattomuus johtuu background-attachment:fixed-tyylimäärityksen implementaation puutteesta näissä selaimissa (24). Lisäksi background-attachment:fixed-määrityksen on todettu aiheuttavan ongelmia IOS Safari-selaimilla käytettäessä yhdessä background-size:cover-määrityksen kanssa (25). Background-attachment:fixed-määritystä vastaava efekti voidaan kuitenkin saavuttaa näillä mobiiliselaimilla noudattamalla tämän alaluvun optimointiohjeita, mutta mobiiliselaimilla nämä määritykset aiheuttavat paint-prosesseja sivustoa vierittäessä.

Näistä mobiiliselaimista aiheutuvien ongelmien vuoksi tässä alaluvussa esiteltävät timeline-nauhoitukset on nauhoitettu kannettavalla Apple Macbook Pro -tietokoneella.

6.3.1 Ongelman kohdentaminen

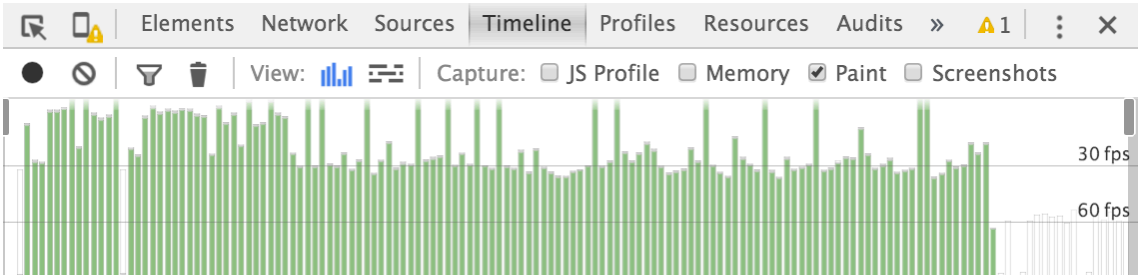
Esimerkkiprojektin Scrolling-näkymässä (http://localhost:9001/#/scrolling) (kuva 18) on nähtävissä kymmenen täytetekstiä sisältävää p-elementtiä sekä Animations-näkymän form-elementtiä vastaava form-elementti, josta käyttäjä voi kytkeä päälle renderöintimoottorille laskennallisesti raskaita tyylejä p-elementeille sekä kytkeä päälle sivustoa optimoivat tyylimääritykset.



Kuva 18. Optimoimaton Scrolling-näkymä

Ensisilmäyksellä Scrolling-näkymässä ei ole varsinaisesti mitään, mikä aiheuttaisi ylimääräisiä paint-prosesseja. Näkymässä ei ole animoituja elementtejä, jotka vaatisivat renderöintimoottoria suorittamaan paint-prosesseja. Aktivoimalla Enable paint flashing-moodi sekä vierittämällä näkymää ylös ja alas voidaan kuitenkin todeta, että renderöintimoottori itse asiassa suorittaa paint-prosesseja koko ruudulle vierityksen aikana. Tämä voidaan todentaa nauhoittamalla timeline-nauhoite, jonka aikana käyttäjä vierittää näkymää edestakaisin (kuva 19). Tällaista koko näytön kattavaa paint-prosessien sarjaa kutsutaan yleisesti termillä *Paint Storm,* ja se aiheuttaa merkittäviä ongelmia ruudunpäivitysnopeuden suhteen myös keskimääräistä tehokkaammilla pöytäkoneilla ja kannettavilla tietokoneilla. (26; 27; 28.)

Nämä ongelmat korostuvat, kun näkymän form-elementistä aktivoi Expensive styling -valinnan, joka lisää p-elementeille laskennallisesti raskaita tyylejä.



Kuva 19. Timeline-nauhoite optimoimattomasta vierityksestä

Koska Scrolling-näkymän esimerkkitapauksessa paint-prosessit peittävät koko näkymän, paint-prosesseja aiheuttavaa elementtiä on mahdotonta yksilöidä manipuloimatta DOM-puuta. Tällaisissa tapauksissa voidaan kuitenkin tehdä olettamus, etteivät paint-prosessit aiheudu elementeistä, jotka ovat kooltaan pienempiä kuin koko näkymä. Tämä olettamus sulkee pois Scrolling-näkymän tapauksessa sekä form-elementin sekä täytetekstit sisältävän div-elementin.

DOM-puuta ylöspäin mentäessä näiden elementtien ensimmäinen vanhempi div-elementti on ensimmäinen koko näkymän kokoinen elementti ja siten järkevä piste aloittaa optimointiprosessi. Elementin piilottaminen Elements-välilehdellä visibility:hidden-määrityksellä piilottaa myös sen sisältämät, käytännössä koko näkymän muodostavat elementit, eikä sen piiloittamisella näin ollen voida saavuttaa optimoimatonta näkymää muistuttavaa näkymää. Sen sijaan tässä tapauksessa on järkevämpää Elements-välilehdellä poistaa elementiltä tyylimäärityksiä yksi kerrallaan Enable paint flashing valinnan ollessa päälle kytkettynä. Kyseinen elementti saa unoptimized-luokalta kuvan 20 mukaiset määritykset.

```
&.unoptimized {
  background: url('images/background.jpg') repeat center center;
  background-attachment: fixed;
  background-size: cover;
}
```

Kuva 20. Scrolling-näkymän optimoimattomat tyylimääritykset

Background-ominaisuuden poistaminen elementin tyylimäärityksistä poistaa myös *Paint Storm* -efektin, mutta samalla katoaa näkymän taustakuva. Web-kehittäjällä on hyvin harvoin mahdollisuus poiketa näin radikaalilla tavalla visuaalisista suunnitelmista tai esimerkiksi asiakkaan toiveista sivuston ulkoasuun liittyen, joten tällainen ratkaisu ei

lähtökohtaisesti ole hyväksyttävissä. Background-size-ominaisuuden poistaminen muuttaa taustakuvan asemoinnin näytön suhteen oletusarvoiseen *auto*-arvoon, eikä poista näkymää vierittäessä ilmeneviä *Paint Storm* -ongelmia.

Background-attachment-ominaisuus muuttaa taustakuvan käyttäytymistä sivustoa vierittäessä. Ominaisuuden poistaminen asettaa background-attachment-ominaisuuden oletusarvoon *scroll*, jonka jälkeen sivuston vierittäminen enable paint flashing -asetus päällä ei aiheuta renderöintimoottorille paint-prosesseja. Samalla menetetään kuitenkin background-attachment:fixed-määrityksellä alun perin tavoiteltu efekti, ruudun suhteen staattisesti asemoituva taustakuva. Tämä määritys voidaan kuitenkin korvata monilla eri tavoilla, joista ehkä yksinkertaisin esitellään seuraavassa alaotsikossa.

6.3.2 Kohdennettujen elementtien optimointi

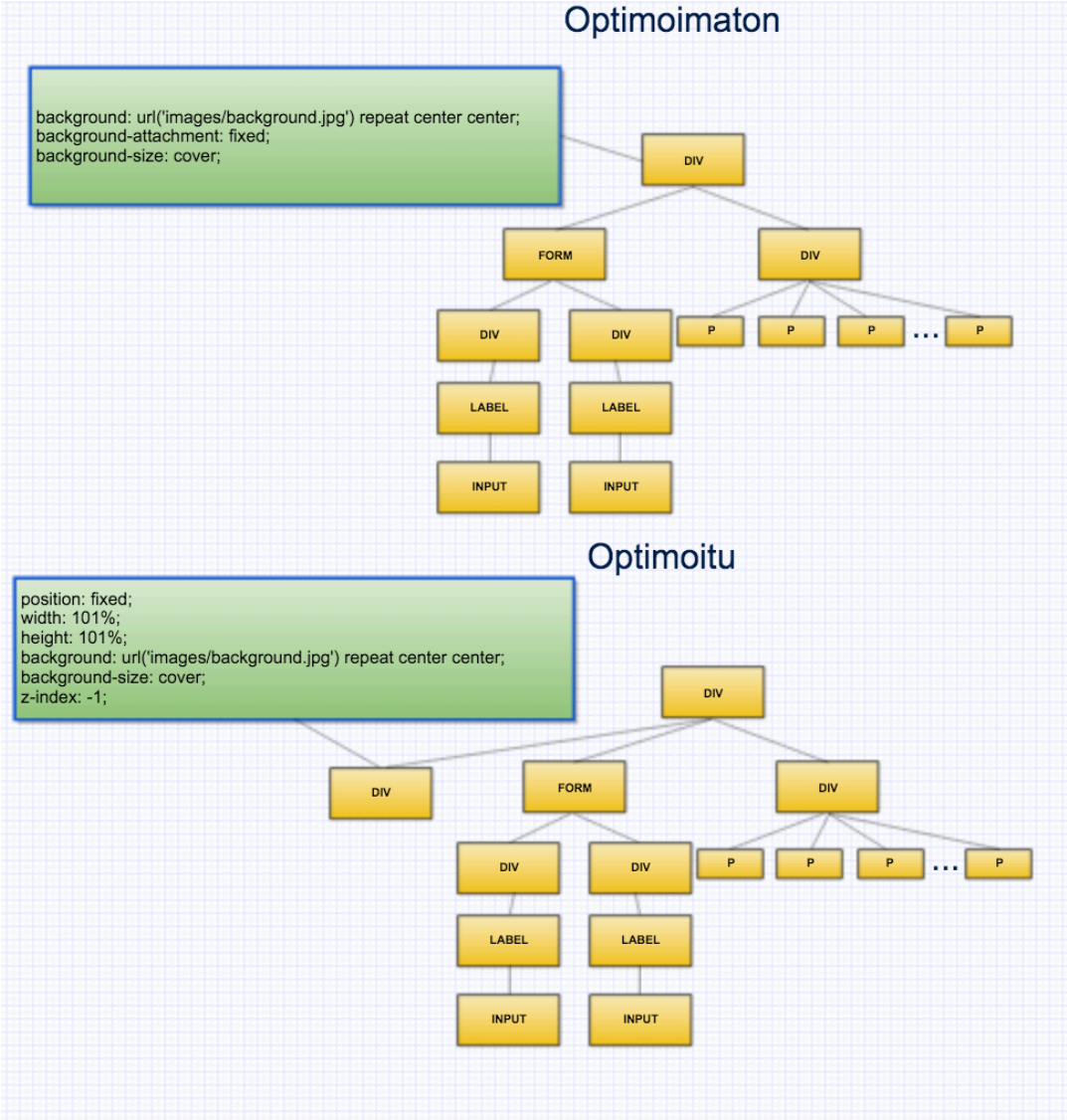
Ruudun suhteen staattisesti asemoituvan taustakuvan luomiseksi on yleisesti käytetty background-attachment:fixed-määritystä, joka aiheuttaa suuria määriä ruudun kokoisia paint-prosesseja sivustoa vierittäessä. Tavanomaisin tapa päästä eroon tarpeettomista paint-prosesseista on niin kutsuttu *null transform hack* (29; 30), jossa paint-prosesseja aiheuttavalle elementille asetetaan elementtiä z-akselilla siirtävä translate: transformZ()-määritys, välittäen transform-funktiolle arvon 0px. Tämä ei siirrä todellisuudessa elementtiä lainkaan, mutta renderöintimoottorin näkökulmasta elementti saattaa siirtyä, joten renderöintimoottori korottaa sen omalle tasolleen grafiikkayksikön renderöitäväksi poistaen tarpeen suorittaa sille paint-prosesseja.

Tämä ei kuitenkaan onnistu suoraan elementille, johon on määritetty background-attachment:fixed-määritys, sillä taustakuvan staattinen asemointi hajoaa *null transform hackia* käytettäessä samassa elementissä. Taustakuvan optimointi voidaan kuitenkin toteuttaa siirtämällä taustakuva-määritykset erilliseen position:static-määrityksellä asemoituun uuteen div-elementtiin, joka määritetään koko näkymän kokoiseksi sekä piirtymään alkuperäisen scrolling-view-luokan saavan div-elementin taakse. Uuden taustakuvasta vastaavan elementin lisäämisessä DOM-puuhun on vain yksi ehto: uuden elementin on oltava optimoimattomasta taustakuvasta vastanneen div-elementin välitön lapsielementti. Uudelle elementille annetaan kuvan 21 mukaiset tyylimääritykset.

```
.background {
  position: fixed;
  width: 101%;
  height: 101%;
  background: url('images/background.jpg') repeat center center;
  background-size: cover;
  z-index: -1;
}
```

Kuva 21. Optimoidusta staattisesta taustakuvasta vastaavan div-elementin tyylimäärittelyt

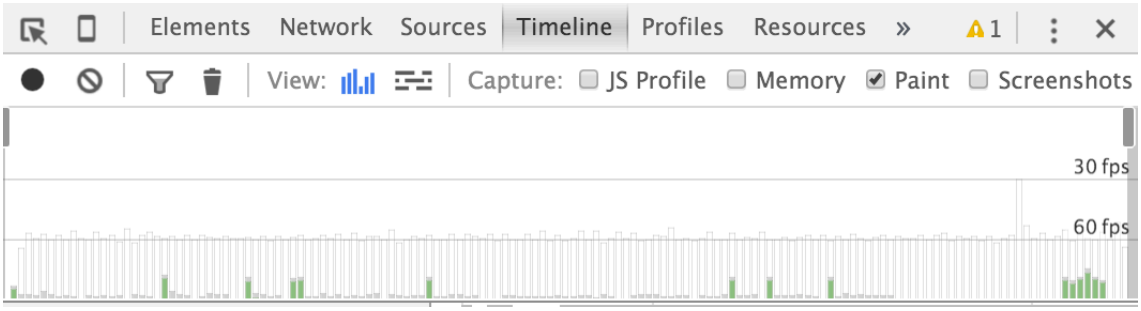
Uusista tyylimäärittelyistä position:fixed korvaa background-attachment:fixed-määrityksen asemoiden div-elementin staattiseksi ruudun suhteen. Asettamalla elementin kooksi yli 100 prosenttia vanhempi-elementistään voidaan olla varmoja, ettei taustakuvan koko pikseleinä pyöristy ikinä pikseliäkään vanhempi-elementin kokoa pienemmäksi, kun renderöintimoottori muuttaa elementin mittasuhteet suhteellisista mittasuhteista pikseleiksi. Asettamalla elementille z-index-ominaisuus negatiivisella arvolla voidaan varmistua siitä, että elementti renderöidään sisarelementtiensä taakse, jolloin taustakuvasta vastaava div-elementti ei peitä näkymän form-elementtiä tai täytetekstiä sisältäviä p-elementtejä. Kuva 22 havainnollistaa optimoimattoman ja optimoidun renderöintipuun välisiä eroja. Selkeyden vuoksi kuvassa esitetään renderöintipuusta vain ne tyylimäärittelyt, jotka muuttuvat optimoinnin yhteydessä.



Kuva 22. Optimoimattoman ja optimoidun renderöintipuun väliset erot

Esimerkkiprojektin Scrolling-näkymässä staattisen taustakuvan optimointi voidaan kytkeä päälle aktivoimalla Optimized scrolling -valinta form-elementistä.

Nauhoittamalla scrolling-näkymästä timeline-nauhoite (kuva 23), jonka aikana käyttäjä vierittää näkymää ylösalaisin, voidaan todeta *Paint Storm* -efektin poistuneen ja renderöintimoottorin suoriutuvan tehtävästään riittävän hyvin tasaisen 60 ruudun ruudunpäivitysnopeuden saavuttamiseksi.



Kuva 23. Timeline-nauhoite optimoidusta vierityksestä

7 Tulevaisuuden näkymät

7.1 Will-change-ominaisuus

Will-change-ominaisuuden on tarkoitus tulevaisuudessa korvata kaikki niin sanotusti semanttisesti väärät keinot korottaa elementin renderöinti grafiikkayksikön renderöitäväksi. Tässä alaluvussa esitellään esimerkkitapaus uuden ominaisuuden käytöstä. Will-change-ominaisuuden merkittävämpänä etuna aiempiin tapoihin korottaa elementin renderöinti omalle tasolleen on elementin korottaminen ennen korotusta vaativien ominaisuuksien aktivoitumista, kuten luvun esimerkissä.

Edellisessä luvussa esitelty null transform hack mahdollistaa ajon aikana muuttuvien elementtien korottamisen omalle tasolleen grafiikkayksikön suoritettavaksi, mutta sen ja muiden korotettuja tasoja generoivien määritysten liiallinen käyttäminen sivustolla saattaa aiheuttaa vakavia suorituskykyongelmia renderöinnin yhteydessä. Huhtikuussa 2014 julkaistussa CSS-työvedoksessa (31) esitelty will-change-ominaisuus pyrkii tarjoamaan CSS-syntaksiin tavan korottaa elementtejä omille tasoilleen ennakoiden tulevaa muutosta niiden ominaisuuksissa sekä korvaamaan semanttisesti validimmalla ominaisuudella yleisesti elementtien korottamiseen käytetyn transform: translateZ()-määrityksen, jonka alkuperäinen funktio on siirtää elementtejä Z-akselin suuntaisesti.

Ohjaamalla kehittäjät käyttämään uutta will-change-ominaisuutta selaimien kehittäjät voivat optimoida renderöintimoottoreitaan käsittelemään will-change-, ja transform: translate-ominaisuuksia siinä olettamuksessa, että niitä käytetään niiden semanttisia tarkoitusperiä mukaillen. Insinöörityön kirjoitushetkellä will-change-ominaisuus on jo melko kattavasti implementoitu moderneihin selaimiin (Firefox 45, Chrome 49, Safari

9.1, Opera 36, Android Browser 47, Chrome for android 49). Implementaatio puuttuu vielä Microsoftin selaimista (IE 11, Edge 13) sekä osasta mobiiliselaimia (iOS Safari 9.2, Opera Mini 8).

7.2 Esimerkki will-change-ominaisuuden käytöstä

Esimerkkiprojektin will-change-näkymässä (http://localhost:9001/#/willchange) esitellään kaksi sisäkkäistä div-elementtiä, joista contain-luokan tyylimäärittelyt saava elementti kapseloi sisäänsä animated-luokan määrittelyt saavan div-elementin. Esimerkin tarkoitus on kuvata uuden will-change-ominaisuuden käyttötarkoitusta. Kuvassa 24 esitetään html-koodi, josta will-change-näkymä rakennetaan.

```
<div class="contain">
  Container
  <div class="animated">
    Animated on hover
  </div>
</div>
```

Kuva 24. Will-change-näkymän html-dokumentti

Esimerkin lapsi-elementtiä on tarkoitus animoida sivuttaissuunnassa käyttämällä transform:translateX-ominaisuutta, kun kursori viedään sen päälle näkymässä. Käyttämällä will-change-ominaisuutta voidaan korottaa animoitava elementti omalle tasolleen. Elementin korottaminen vaatii kuitenkin renderöintimoottorilta aikaa, jonka vuoksi will-change-määritys suositellaan (32; 33) asetettavan elementille esimerkiksi siinä vaiheessa, kun hiiri viedään sen välittömän isäelementin päälle. Esimerkkitapauksessa tämä elementti on Contain-luokan omaava elementti.

Kuvassa 25 esitetään LESS-syntaksilla (34) kuvatut tyylimäärittelyt näkymän molemmille elementeille.

```
.contain {
  width: 50%;
  margin: 100px auto;
  padding-top: 10%;
  background-color: #DDDDDD;
  height: 200px;
  text-align: center;
  &:hover {
    .animated {
      will-change: transform;
    }
  }
  .animated {
    width: 50%;
    height: 50%;
    position: relative;
    margin: auto;
    background-color: #AAAAAA;
    &:hover {
      animation: move 2s linear infinite;
    }
  }
```

Kuva 25. Will-change-näkymän elementtien tyylimäärittelyt

Will-change-ominaisuuden toiminnallisuutta voidaan tutkia kytkemällä show layer border -asetus päälle Chrome devToolsissa. Esimerkkiprojektin will-change-näkymässä kursorin vieminen vanhempi-elementin päälle korottaa lapsi-elementin omalle tasolleen grafiikkayksikön renderöitäväksi, jolloin renderöintimoottori maalaa lapsi-elementin reunat oranssilla värillä.

8 Yhteenveto

Insinöörityön tarkoituksena oli esittää mahdollisimman ymmärrettävästi selaimen renderöintimoottorin toimintaa, heikon suorituskyvyn omaavien web-sivujen optimointiprosessia, sekä joitakin tavanomaisimpia web-sivuston heikon suorityskyvyn aiheuttavia ominaisuuksia. Insinöörityö soveltuu luettavasti kaikille web-kehittäjille, jotka ovat kiinnostuneita rakentamaan sivustoilleen transitioita, animaatioita tai muita esteettisiä muotoiluja tavalla, joka ei rajoita sivuston suorituskykyä. Yhtenä jatkokehitysmahdollisuute-

na työlle näen optimointiprosessin käsittelyn selaimessa suoritettavien JavaScript-skriptien näkökulmasta.

Työn aikana ilmeni haasteita, jotka liittyvät pääasiassa modernien selaimien nopeaan kehittymiseen. Työssä kuvatut optimointiongelmat ovat jossain määrin selainkohtaisia ja selainkehittäjät tekevät jatkuvasti työtä näiden ongelmien ratkaisemiseksi renderöintimoottorien kuorten alla. Lisäksi selainkehittäjät kehittävät jatkuvasti optimointiprosessissa käytettyjä työkaluja, mikä ilmeni esimerkiksi kirjoittamisen aikana päivittyneen Google Chrome -selaimen devTools-työkalun visuaalisen ilmeen ja ominaisuuksien muuttumisena eri selainversioissa. Toinen kirjoittamisen aikana ilmennyt selainten kehityssykliin liittyvä haaste oli lähdemateriaalin niukkuus ja ajankohtaisuus. Renderöintimoottorien optimointiin liittyvä materiaali on pääsääntöisesti muutamien aktiivisten selainkehittäjien kirjoittamia ja käsittelee ongelmia, joiden ratkaisun selainkehittäjät pyrkivät tulevissa selainversioissa automatisoimaan renderöintimoottorin implementaatiossa.

Olen tyytyväinen siihen, millä tarkkuudella olen pystynyt kuvaamaan web-kehityksen osa-aluetta, jonka tuntemus on mielestäni keskimääräisesti suhteellisen heikolla tasolla web-kehittäjien keskuudessa. Insinöörityön lukemalla saa mielestäni hyvän käsityksen ruudunpäivitystaajuuteen vaikuttavista ongelmista sekä niiden optimoinnista. Työtä kirjoittaessa kasvatin merkittävästi omaa osaamistani aiheesta, mikä oli tavoitteena myös työn tilaajan, Tieto CEM:n, näkökulmasta.

Lähteet

1. Browser Statistics. 2016. Verkkodokumentti. <http://www.w3schools.com/browsers/browsers_stats.asp>. luettu 11.4.2016.

2. Garsiel T, Irish P. 2011. How Browsers Work: Behind the scenes of modern web browsers. Verkkodokumentti. <http://www.html5rocks.com/en/tutorials/internals/howbrowserswork/>. luettu 10.10.2015.

3. Top 12 Desktop, Tablet & Console Browser Versions on May 2012 | StatCounter Global Stats. 2012. Verkkodokumentti. <http://gs.statcounter.com/#browser_version-ww-monthly-201205-201205-bar>. luettu 15.10.2015.

4. Edge Conference - Panel 3: Performance. 2013. Verkkodokumentti. <https://www.youtube.com/watch?v=3-WYu_p5rdU>. luettu 4.11.2015.

5. Grigorik I. 2015. Constructing the Object Model Verkkodokumentti. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model>. luettu 5.12.2015.

6. Grigorik I. 2015. Render-tree construction, layout, and paint Verkkodokumentti. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction>. luettu 8.12.2015. Käyttölisenssi: Creative Commons Attribution 3.0 License <http://creativecommons.org/licenses/by/3.0/>

7. Singh S. 2013. Compositing in Blink and WebKit. Verkkodokumentti. <https://www.youtube.com/watch?v=Lpk1dYdo62o>. luettu 21.11.2015.

8. Visual formatting model. Verkkodokumentti. <https://www.w3.org/TR/CSS21/visuren.html#z-index>. luettu 15.10.2015.

9. Osmani A. 2012. Improving Web App Performance With the Chrome DevTools Timeline and Profiles. Verkkodokumentti. <https://addyosmani.com/blog/performance-optimisation-with-timeline-profiles/ >. luettu 23.11.2015.

10. Performance profiling with the Timeline - Google Chrome. Verkkodokumentti. <https://developer.chrome.com/devtools/docs/timeline#frames_mode>. luettu 11.1.2016.

11. Irish P. 2015. Profiling Long Paint Times with DevTools' Continuous Painting Mode. Verkkodokumentti. <https://developers.google.com/web/updates/2013/02/Profiling-Long-Paint-Times-with-DevTools-Continuous-Painting-Mode>. luettu 12.11.2015.

12. Osmani A. 2013. Gone In 60 Frames Per Second: A Pinterest Paint Performance Case Study – Smashing Magazine. Verkkodokumentti. <https://www.smashingmagazine.com/2013/06/pinterest-paint-performance-case-study/>. luettu 22.11.2015.

13. Lewis P. 2013. A Rendering Performance Guide for Developers by Paul Lewis (#perfmatters at SFHTML5). Verkkodokumentti. <https://www.youtube.com/watch?v=9xjpmpX4NJE>. luettu 27.11.2015.

14. Lewis P, Irish P. 2013. High Performance Animations. Verkkodokumentti. <http://www.html5rocks.com/en/tutorials/speed/high-performance-animations/>. luettu 11.12.2015.

15. CSS properties by style operation required. 2016. Verkkodokumentti. <https://docs.google.com/a/tieto.com/spreadsheets/d/1Hvi0nu2wG3oQ51XRHtMv-A_ZlidnwUYwgQsPQUg1R2s/pub?single=true&gid=0&output=html>. luettu 13.2.2016.

16. Lewis P. ; 2016. CSS Triggers. Verkkodokumentti. <https://csstriggers.com/>. luettu 22.3.2016.

17. Node.js Foundation. ; 2016. Node.js. Verkkodokumentti. <https://nodejs.org>. luettu 12.3.2016.

18. 2016. Grunt: The JavaScript Task Runner. Verkkodokumentti. <http://gruntjs.com/>. luettu 15.3.2016.

19. 2016. npm. Verkkodokumentti. <https://www.npmjs.com/>. luettu 20.3.2016.

20. 2016. Bower. Verkkodokumentti. <http://bower.io/>. luettu 19.3.2016.

21. 2016. Git. Verkkodokumentti. <https://git-scm.com/>. luettu 11.3.2016.

22. Jackson D, Hyatt D, Marrin C. 2009. CSS 3D Transforms Module Level 3. Verkkodokumentti. <https://www.w3.org/TR/2009/WD-css3-3d-transforms-20090320/>. luettu 2.4.2016.

23. 2015. Scrolling performance monitoring with Chrome Devtools - transparent bars. Verkkodokumentti. <http://stackoverflow.com/questions/23229949/scrolling-performance-monitoring-with-chrome-devtools-transparent-bars/30395602#30395602>. luettu 27.3.2016.

24. Caniuse - background-attachment. 2016. Verkkodokumentti. <http://caniuse.com/#search=background-attachment>. luettu 5.4.2016.

25. html - Background size on iOS. 2015. Verkkodokumentti. <http://stackoverflow.com/questions/21476380/background-size-on-ios>. luettu 26.3.2016.

26. Lewis P. 2013. Performance Calendar » The Runtime Performance Checklist. Verkkodokumentti. <http://calendar.perfplanet.com/2013/the-runtime-performance-

checklist/>. luettu 3.4.2016.

27. Lewis P. Chrome Dev Summit: Performance Summary | Web Updates. Verkkodokumentti. <https://developers.google.com/web/updates/2014/01/Chrome-Dev-Summit-Performance-Summary>. luettu 27.3.2016.

28. Bithrey W. 2013. Jank Busting - Fixed Backgrounds. Verkkodokumentti. <http://rathersemantic.com/2013/08/22/jank-busting-fixed-backgrounds-ea/>. luettu 27.2.2016.

29. Irish P. 2013. Aerotwist - On translate3d and layer creation hacks. Verkkodokumentti. <https://aerotwist.com/blog/on-translate3d-and-layer-creation-hacks/>. luettu 13.3.2016.

30. Osmani A. 2013. Take Care When Using Null Transform Hacks For Forcing GPU Acceleration. Verkkodokumentti. <https://addyosmani.com/blog/be-careful-when-using-null-transform-hacks-to-force-gpu-acceleration/>. luettu 27.3.2016.

31. Atkins TJ. 2014. CSS Will Change Module Level 1. Verkkodokumentti. <https://www.w3.org/TR/2014/WD-css-will-change-1-20140429/>. luettu 28.3.2016.

32. Soueidan S. 2014. Everything You Need to Know About the CSS will-change Property. Verkkodokumentti. <https://dev.opera.com/articles/css-will-change-property/>. luettu 27.3.2016.

33. Atkins TJ. 2016. CSS Will Change Module Level 1. Verkkodokumentti. <https://drafts.csswg.org/css-will-change/>. luettu 21.3.2016.

34. Less.js. 2016. Verkkodokumentti. <http://lesscss.org/>. luettu 12.3.2016.